Informix Guide to SQL

Syntax

Version 7.2
April 1996
Part No. 000-7859A

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by "®," and in numerous other countries worldwide:

INFORMIX®; C-ISAM®; INFORMIX®-OnLine Dynamic Server™

The following are worldwide trademarks of the indicated owners or their subsidiaries, registered in the United States of America as indicated by "®," and in numerous other countries worldwide:

Regents of the University of California: BSD™
Microsoft Corporation: Microsoft®; MS®; MS-DOS®;
("DOS" as used herein refers to MS-DOS and/or PC-DOS operating systems.)
X/OpenCompany Ltd.: UNIX®; X/Open®

Some of the products or services mentioned in this document are provided by companies other than Informix. These products or services are identified by the trademark or servicemark of the appropriate company. If you have a question about one of those products or services, please call the company in question directly.

Documentation Team: Diana Chase, Geeta Karmarkar, Dawn Maneval, Tom Noronha, Patrice O'Neill

Copyright © 1981-1996 by Informix Software, Inc. All rights reserved.

No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

To the extent that this software allows the user to store, display, and otherwise manipulate various forms of data, including, without limitation, multimedia content such as photographs, movies, music and other binary large objects (blobs), use of any single blob may potentially infringe upon numerous different third-party intellectual and/or proprietary rights. It is the user's responsibility to avoid infringements of any such third-party rights.

RESTRICTED RIGHTS LEGEND

Software and accompanying materials acquired with United States Federal Government funds or intended for use within or for any United States federal agency are provided with "Restricted Rights" as defined in DFARS 252.227-7013(c)(1)(ii) or FAR 52.227-19.

Table of Contents

Introduction

Products Covered in This Manual 3
About This Manual 3
- Organization of This Manual 4
- Types of Users 5
- Software Dependencies 6
- Demonstration Database 8

New Features of This Product 11
Conventions 16
- Typographical Conventions 16
- Icon Conventions 17
- Syntax Conventions 19
- Sample-Code Conventions 24
- Terminology Conventions 25

Additional Documentation 26
- Printed Documentation 26
- On-Line Documentation 27
- Related Reading 29

Compliance with Industry Standards 30
Informix Welcomes Your Comments 30

Chapter 1 SQL Statements

How to Enter SQL Statements 1-6
How to Enter SQL Comments 1-9
Categories of SQL Statements 1-12
ANSI Compliance and Extensions 1-16
Statements 1-18
- ALLOCATE DESCRIPTOR 1-19
- ALTER FRAGMENT 1-22
- ALTER INDEX 1-43
- ALTER TABLE 1-46
- BEGIN WORK 1-77

CHECK TABLE 1-79
CLOSE 1-81
CLOSE DATABASE 1-85
COMMIT WORK. 1-87
CONNECT 1-89
CREATE AUDIT 1-102
CREATE DATABASE 1-104
CREATE INDEX 1-109
CREATE PROCEDURE 1-134
CREATE PROCEDURE FROM 1-144
CREATE ROLE 1-145
CREATE SCHEMA 1-147
CREATE SYNONYM 1-150
CREATE TABLE 1-154
CREATE TRIGGER 1-192
CREATE VIEW 1-224
DATABASE. 1-229
DEALLOCATE DESCRIPTOR 1-232
DECLARE 1-234
DELETE 1-252
DESCRIBE 1-255
DISCONNECT 1-261
DROP AUDIT 1-265
DROP DATABASE 1-266
DROP INDEX 1-268
DROP PROCEDURE 1-270
DROP ROLE 1-271
DROP SYNONYM 1-272
DROP TABLE 1-274
DROP TRIGGER 1-277
DROP VIEW 1-279
EXECUTE 1-281
EXECUTE IMMEDIATE 1-290
EXECUTE PROCEDURE 1-293
FETCH 1-296
FLUSH 1-308
FREE 1-311
GET DESCRIPTOR 1-314
GET DIAGNOSTICS 1-321
GRANT 1-340
GRANT FRAGMENT 1-356
INFO 1-365
INSERT 1-370
LOAD 1-380
LOCK TABLE 1-387
OPEN. 1-390
OUTPUT 1-400
PREPARE 1-402
PUT 1-416

RECOVER TABLE 1-425
RENAME COLUMN 1-428
RENAME DATABASE 1-431
RENAME TABLE 1-432
REPAIR TABLE 1-435
REVOKE 1-437
REVOKE FRAGMENT 1-450
ROLLBACK WORK 1-455
ROLLFORWARD DATABASE. 1-457
SELECT 1-459
SET 1-501
SET CONNECTION 1-527
SET DATASKIP 1-534
SET DEBUG FILE TO 1-537
SET DESCRIPTOR 1-540
SET EXPLAIN 1-548
SET ISOLATION 1-556
SET LOCK MODE 1-561
SET LOG 1-564
SET OPTIMIZATION 1-566
SET PDQPRIORITY 1-568
SET ROLE 1-571
SET SESSION AUTHORIZATION 1-573
SET TRANSACTION 1-575
START DATABASE 1-581
START VIOLATIONS TABLE 1-584
STOP VIOLATIONS TABLE 1-603
UNLOAD. 1-605
UNLOCK TABLE 1-610
UPDATE 1-612
UPDATE STATISTICS 1-623
WHENEVER. 1-632
Segments 1-640
Condition 1-643
Constraint Name 1-658
Database Name 1-660
Data Type 1-664
DATETIME Field Qualifier 1-669
Expression 1-671
Identifier 1-723
Index Name 1-741
INTERVAL Field Qualifier 1-743
Literal DATETIME. 1-746
Literal INTERVAL 1-749
Literal Number 1-752
Procedure Name 1-754
Quoted String 1-757
Relational Operator 1-761
Synonym Name 1-766

Table Name 1-768
View Name 1-772

Chapter 2 SPL Statements

CALL 2-4
CONTINUE 2-7
DEFINE 2-8
EXIT 2-16
FOR 2-18
FOREACH 2-23
IF 2-27
LET 2-31
ON EXCEPTION 2-34
RAISE EXCEPTION 2-39
RETURN 2-41
SYSTEM 2-43
TRACE 2-46
WHILE 2-49

Index

Introduction

Products Covered in This Manual 3

About This Manual 3
- Organization of This Manual 4
- Types of Users 5
- Software Dependencies 6
  - Assumptions About Your Database Server 6
  - Assumptions About Your Locale 7
- Demonstration Database 8

New Features of This Product 11

Conventions 16
- Typographical Conventions 16
- Icon Conventions 17
  - Comment Icons 17
  - Product and Platform Icons 18
  - Compliance Icons 19
- Syntax Conventions 19
  - Elements That Can Appear on the Path 20
  - How to Read a Syntax Diagram 23
- Sample-Code Conventions 24
- Terminology Conventions 25
  - Definitions of Terms 25
  - Abbreviations of Product Names 25

Additional Documentation 26
Printed Documentation 26
On-Line Documentation 27
Error Message Files 28
Release Notes, Documentation Notes, Machine Notes 28
Related Reading 29

Compliance with Industry Standards. 30

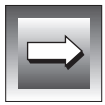
Informix Welcomes Your Comments 30

This chapter introduces the *Informix Guide to SQL: Syntax* manual. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout this manual.

Products Covered in This Manual

All the information presented in this manual is valid for the following products. Differences in their use of SQL are indicated where appropriate:

- INFORMIX-ESQL / C, Version 7.2
- INFORMIX-ESQL / COBOL, Version 7.2
- INFORMIX-OnLine Dynamic Server, Version 7.2
- INFORMIX-SE, Version 7.2



Important: *This manual does not cover the product called INFORMIX-SQL or any other Informix application development tool. Each of these tools is covered in its own manual set. This manual is valid only for the products in the preceding list.*

About This Manual

The *Informix Guide to SQL: Syntax* is intended to be used as a companion volume to the *Informix Guide to SQL: Reference* and *Informix Guide to SQL: Tutorial*. This manual and the *Informix Guide to SQL: Reference* are references that you can use on a daily basis after you finish reading and experimenting with the *Informix Guide to SQL: Tutorial*.

This manual contains all the syntax descriptions for Structured Query Language (SQL) and Stored Procedure Language (SPL) statements. The *Informix Guide to SQL: Tutorial* explains the philosophy and concepts behind relational databases, and the *Informix Guide to SQL: Reference* provides reference information for aspects of SQL other than the language statements.

Organization of This Manual

The *Informix Guide to SQL: Syntax* includes the following chapters:

- The Introduction explains the purpose and organization of this manual, introduces the demonstration database from which the product examples in the manual are drawn, lists the new features for Version 7.2 of Informix database server products that use SQL, and describes the syntax conventions followed in the manual.
- Chapter 1, "SQL Statements," describes SQL statements and segments. The chapter is divided into six sections. The first four sections provide an introduction to the statements and segments. These sections cover the following subjects: entry of SQL statements, entry of SQL comments, categories of SQL statements, and categories of ANSI compliance. The fifth and sixth sections, "Statements" and "Segments," are the major sections of the chapter.
  - "Statements" explains the workings of all the SQL statements that Informix products support. Detailed syntax diagrams walk you through every clause of each SQL statement, and syntax tables explain the input parameters for each clause. Thorough usage instructions, pertinent examples, and references to related material complete the description of each SQL statement.
  - "Segments" explains all the SQL segments. SQL segments are language elements, such as table names and expressions, that occur in many SQL statements. Instead of describing each segment in each statement where it occurs, this manual provides a comprehensive stand-alone description of each segment. Whenever a segment occurs within a given syntax diagram, the diagram points to the stand-alone description of the segment in this section for further information.

- Chapter 2, “SPL Statements,” presents all the detailed syntax diagrams and explanations for SPL statements. You can use stored procedures to perform any function you can perform in SQL as well as to expand what you can accomplish with SQL alone. You write a stored procedure using SPL and SQL statements. For task-oriented information about using stored procedures, see the *Informix Guide to SQL: Tutorial*.
- The Index is a combined index for the manuals in the SQL series. Each page reference in the index ends with a code that identifies the manual in which the page appears. The same index also appears in the *Informix Guide to SQL: Reference* and the *Informix Guide to SQL: Tutorial*.

The following items in the SQL manual series are an integral part of the *Informix Guide to SQL: Syntax* even though they do not appear physically in this manual:

- A description of the **stores7** database appears in Appendix A of the *Informix Guide to SQL: Reference*. This appendix describes the structure and contents of the demonstration database that is installed with the Informix database server products. All the manuals in the SQL manual series use this database for their examples.
- Definitions of terms that are used in the SQL manual series appear in the Glossary of the *Informix Guide to SQL: Reference*.

SQL messages that the database server issues to reflect errors in the execution of SQL statements are described in the section on SQL error codes in the *Informix Error Messages* manual.

Types of Users

This manual is written for people who use Informix products and SQL on a regular basis. The primary audience for this manual consists of SQL developers and database administrators. The secondary audience consists of end users and anyone else who needs to know the syntax of SQL statements.

Software Dependencies

You must have the following Informix software to enter and execute SQL and SPL statements:

- An INFORMIX-OnLine Dynamic Server database server or an INFORMIX-SE database server.

  The database server must be installed either on your computer or on another computer to which your computer is connected over a network.

- Either an Informix application development tool, such as INFORMIX-4GL; an SQL application programming interface (API), such as INFORMIX-ESQL/C; or the DB-Access database access utility, which is shipped as part of your database server.

  The application development tool, the SQL API, or DB-Access enables you to compose queries, send them to the database server, and view the results that the database server returns.

  You can use DB-Access to try out many of the SQL statements described in this manual. See the *DB-Access User Manual* for a list of all the SQL statements that you can run from DB-Access.

Assumptions About Your Database Server

This manual assumes that you are using INFORMIX-OnLine Dynamic Server as your database server. Thus information that is valid only for OnLine is not called out as such in the text of this manual, unless a particular piece of text emphasizes a contrast between OnLine and INFORMIX-SE. However, in syntax diagrams, syntax paths valid only for OnLine are identified by a product icon as an aid to users.

Features and behavior specific to INFORMIX-SE are noted throughout the text of this manual. Information valid only for SE is identified by a product icon in the text, and syntax paths valid only for SE are identified by the same product icon in syntax diagrams.

Assumptions About Your Locale

This manual assumes that you are using the default locale, U.S. 8859-1 English. This locale provides support for the U.S. English language, the ISO 8859-1 code set, and standard U.S. conventions for the formatting of monetary, numeric, date, and time data. This locale has the following name:

```
en_us.8859-1
```

In this name, **en** indicates the English language, **us** indicates the United States territory, and **8859-1** indicates the ISO 8859-1 code set. The ASCII code set is a subset of the ISO 8859-1 code set.

However, Informix products can support more than one language, culture, or code set. All culture-specific information is brought together in a single environment, called a GLS (Global Language Support) locale. If you plan to use non-English characters in your data or in SQL identifiers, or if you want to conform to the sorting and collation standards of a non-U.S. English language or territory, you need to specify the appropriate nondefault locale. See the *Guide to GLS Functionality* for instructions on specifying a locale.

If you are using the default U.S. English locale, the *Informix Guide to SQL: Syntax* provides all the syntax information that you need for using SQL statements and segments. You do not need to refer to the *Guide to GLS Functionality* for any additional syntax or considerations.

However, if you are using a nondefault locale, you need to use the *Informix Guide to SQL: Syntax* in conjunction with the *Guide to GLS Functionality*. The chapter on SQL features in that manual explains the effect of the GLS feature on SQL statements and segments. That chapter approaches SQL from a non-U.S. English perspective, so it is a necessary companion to the *Informix Guide to SQL: Syntax* if you are using a nondefault locale.

Demonstration Database

The DB-Access utility, which is provided with your Informix database server products, includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. The sample command files that make up a demonstration application are also included.

Most examples in this manual are based on the **stores7** demonstration database. The **stores7** database is described in detail and its contents are listed in Appendix A of the *Informix Guide to SQL: Reference*.

The script that you use to install the demonstration database is called **dbaccessdemo7** and is located in the **$INFORMIXDIR/bin** directory. The database name that you supply is the name given to the demonstration database. If you do not supply a database name, the name defaults to **stores7**. Use the following rules for naming your database:

- Names can have a maximum of 18 characters for INFORMIX-OnLine Dynamic Server databases and a maximum of 10 characters for INFORMIX-SE databases.
- The first character of a name must be a letter or an underscore (_).
- You can use letters, characters, and underscores (_) for the rest of the name.
- DB-Access makes no distinction between uppercase and lowercase letters.
- The database name must be unique.

When you run **dbaccessdemo7**, as the creator of the database, you are the owner and Database Administrator (DBA) of that database.

If you install your Informix database server according to the installation instructions, the files that constitute the demonstration database are protected so that you cannot make any changes to the original database.

You can run the **dbaccessdemo7** script again whenever you want to work with a fresh demonstration database. The script prompts you when the creation of the database is complete and asks if you would like to copy the sample command files to the current directory. Enter `N` if you have made changes to the sample files and do not want them replaced with the original versions. Enter `Y` if you want to copy over the sample command files.

To create and populate the stores7 demonstration database

1. Set the **INFORMIXDIR** environment variable so that it contains the name of the directory in which your Informix products are installed.
2. Set **INFORMIXSERVER** to the name of the default database server.

   The name of the default database server must exist in the **$INFORMIXDIR/etc/sqlhosts** file. (For a full description of environment variables, see Chapter 4 of the *Informix Guide to SQL: Reference*.) For information about **sqlhosts**, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide* or the *INFORMIX-SE Administrator's Guide*.
3. Create a new directory for the SQL command files. Create the directory by entering the following command:

   ```
   mkdir dirname
   ```
4. Make the new directory the current directory by entering the following command:

   ```
   cd dirname
   ```

5. Create the demonstration database and copy over the sample command files by entering the **dbaccessdemo7** command.

   To create the database without logging, enter the following command:

   ```
   dbaccessdemo7 dbname
   ```

   To create the demonstration database with logging, enter the following command:

   ```
   dbaccessdemo7 -log dbname
   ```

   If you are using INFORMIX-OnLine Dynamic Server, by default the data for the database is put into the root dbspace. If you wish, you can specify a dbspace for the demonstration database.

   To create a demonstration database in a particular dbspace, enter the following command:

   ```
   dbaccessdemo7 dbname -dbspace dbspacename
   ```

   You can specify all of the options in one command, as shown in the following command:

   ```
   dbaccessdemo7 -log dbname -dbspace dbspacename
   ```

   If you are using INFORMIX-SE, a subdirectory called ***dbname*.dbs** is created in your current directory and the database files associated with **stores7** are placed there. You will see both data (**.dat**) and index (**.idx**) files in the ***dbname*.dbs** directory. (If you specify a dbspace name, it is ignored.)

   To use the database and the command files that have been copied to your directory, you must have UNIX read and execute permissions for each directory in the pathname of the directory from which you ran the **dbaccessdemo7** script. Check with your system administrator for more information about operating-system file and directory permissions. UNIX permissions are discussed in the *INFORMIX-OnLine Dynamic Server Administrator's Guide* and the *INFORMIX-SE Administrator's Guide*.

6. To give someone else the permissions to access the command files in your directory, use the UNIX **chmod** command.

7. To give someone else access to the database that you have created, grant them the appropriate privileges using the GRANT statement.

   To revoke privileges, use the REVOKE statement. The GRANT and REVOKE statements are described in Chapter 1 of this manual.

New Features of This Product

The Introduction to each Version 7.2 product manual contains a list of new features for that product. The Introduction to each manual in the Version 7.2 *Informix Guide to SQL* series contains a list of new SQL features.

A comprehensive list of all of the new features for Version 7.2 Informix products is in the release-notes file called **SERVERS_7.2**.

This section highlights the major new features implemented in Version 7.2 of Informix products that use SQL:

- Addition of Global Language Support (GLS)

  The GLS feature allows you to work in any supported language and to conform to the customs of a specific territory by setting certain environment variables. In support of GLS, CHAR and VARCHAR, columns of the system catalog tables are created as NCHAR and NVARCHAR columns in this release. In addition, hidden rows have been added to the **systables** system catalog table. See the discussion of GLS in Chapter 1 of the *Informix Guide to SQL: Reference*.

- ANSI flagger

  The ANSI flagger that Informix products use has been modified to eliminate the flagging of certain SQL items as Informix extensions. These items include the AS keyword in the SELECT clause of the SELECT statement and delimited identifiers in the Identifier segment.

- Bidirectional indexes

  The database server can now traverse an index in either ascending or descending order. So you no longer need to create both an ascending index and a descending index for a column when you use this column in both SELECT...ORDER BY *column name* ASC statements and SELECT...ORDER BY *column name* DESC statements. You only need to create a single ascending or descending index for these queries. See the CREATE INDEX and SELECT statements.

- Column matches in conditions

  When you specify a LIKE or MATCHES condition in the SELECT statement or other statements, you can specify a column name on both sides of the LIKE or MATCHES keyword. The database server retrieves a row when the values of the specified columns match. See the Condition segment and the SELECT statement.

- Column substrings in queries

  You can specify column subscripts for the column named in a SELECT...ORDER BY statement. The database server sorts the query results by the value of the column substring rather than the value of the entire column.

- Column updates after a fetch

  When you use the FOR UPDATE clause of the SELECT statement, you can use the OF *column name* option of this clause to limit the columns that can be updated after a fetch.

- Connectivity information

  You can use the **INFORMIXSQLHOSTS** environment variable to specify the pathname of the file where the client or the database server looks for connectivity information.

- COUNT function

  The ALL *column name* option of the COUNT function returns the total number of non-null values in the specified column or expression. See the Expression segment.

- Data distributions

  You can suppress the construction of index information in the MEDIUM and HIGH modes of the UPDATE STATISTICS statement. When you use the new DISTRIBUTIONS ONLY option of this statement, the database server gathers only distributions information and table information.

- Database renaming

  You can rename local databases. See the new RENAME DATABASE statement.

- DBINFO function

  You can use the 'sessionid' option of the DBINFO function to return the session ID of your current session. See the Expression segment.

- Decimal digits in client applications

  Informix client applications (including the DB-Access utility or any ESQL program that you write) by default display 16 decimal digits of data types FLOAT, SMALLFLOAT, and DECIMAL. The actual digits that are displayed can vary according to the size of the character buffer. The new **DBFLTMASK** environment variable allows you to override the default of 16 decimal digits in the display.

- Default privileges on tables

  You can use the new **NODEFDAC** environment variable to prevent default table privileges from being granted to PUBLIC when a new table is created in a database that is not ANSI compliant.

- Fragment authorization

  You can grant and revoke privileges on individual fragments of tables. See the new GRANT FRAGMENT and REVOKE FRAGMENT statements and the new **sysfragauth** system catalog table.

- High-Performance Loader (HPL) configuration

  You can use the new **DBONPLOAD** and **PLCONFIG** environment variables to specify the names of files and databases to be used by HPL.

- In-place alter algorithm

  INFORMIX-OnLine Dynamic Server uses a new in-place alter algorithm for altering tables when you add a column to the end of the table. See the ALTER TABLE statement.

- Next century in year values

  You can use the next century to expand two-digit year values. See the new **DBCENTURY** environment variable, the Literal DATETIME segment, the DATE data type, and the DATETIME data type.

- Not null constraints

  You can now create not null constraints with the CREATE TABLE and ALTER TABLE statements. The database server records not null constraints in the **sysconstraints** and **syscoldepend** system catalog tables.

- Object modes

  You can specify the object mode of database objects with the new SET statement. This statement permits you to set the object mode of constraints, indexes, and triggers or the transaction mode of constraints. See the SET statement, the new **sysobjstate** system catalog table, and the new syntax for object modes in ALTER TABLE, CREATE INDEX, CREATE TABLE, and CREATE TRIGGER.

- Optical StageBlob area

  You can use the new **INFORMIXOPCACHE** environment variable to specify the size of the memory cache for the Optical StageBlob area of the client application.

- RANGE, STDEV, and VARIANCE functions

  You can use the new aggregate functions RANGE, STDEV, and VARIANCE. See the new syntax for Aggregate Expressions in the Expression segment.

- Roles

  You can create, drop, and enable roles. You can grant roles to individual users and to other roles, and you can grant privileges to roles. You can revoke a role from individual users and from another role, and you can revoke privileges from a role. See the new CREATE ROLE, DROP ROLE, and SET ROLE statements and the new **sysroleauth** system catalog table. Also see the new syntax for roles in the GRANT and REVOKE statements and the new information in the **sysusers** system catalog table.

- Separation of administrative tasks

  The security feature of role separation allows you to separate administrative tasks performed by different groups that are running and auditing OnLine. The **INF_ROLE_SEP** environment variable allows you to implement role separation during installation of OnLine.

- Session authorization

  You can change the user name under which database operations are performed in the current session and thus assume the privileges of the specified user during the session. See the new SET SESSION AUTHORIZATION statement.

- Table access after loads

  The FOR READ ONLY clause of the SELECT statement allows you to access data in the tables of an ANSI-mode database after you have loaded the data with the High-Performance Loader but before you have performed a level-0 backup of the data. After you have performed the level-0 backup, you no longer need to use the FOR READ ONLY clause. See the SELECT and DECLARE statements.

- Thread-safe applications

  You can use the new **THREADLIB** environment variable to compile thread-safe ESQL/C applications. In a thread-safe ESQL/C application, you can use the DORMANT option of the SET CONNECTION statement to make an active connection dormant.

- Tutorials

  Tutorial information on new features has been added to the *Informix Guide to SQL: Tutorial*. The new tutorials cover Global Language Support (GLS), thread-safe applications, object modes, violation detection, fragment authorization, and roles.

- Utilities

  The **dbexport**, **dbimport**, **dbload**, and **dbschema** utilities have been moved from the *Informix Guide to SQL: Reference* to the *Informix Migration Guide*.

- Violation detection

  You can create special tables called violations and diagnostics tables to detect integrity violations. See the new START VIOLATIONS TABLE and STOP VIOLATIONS TABLE statements and the new **sysviolations** system catalog table.

- XPG4 compliance

  SQL statements and data structures have been modified to provide enhanced compliance with the *X/Open Portability Guide 4* (XPG4) specification for SQL. The **sqlwarn** array within the SQL Communications Area (SQLCA) has been modified. A new SQLSTATE code (01007) has been added. The behavior of the ALL keyword in the GRANT statement and the behavior of the ALL and RESTRICT keywords in the REVOKE statement has changed.

See this manual for SQL statements and segments. See the *Informix Guide to SQL: Reference* for data types, system catalog tables, and environment variables. See the *Informix Guide to SQL: Tutorial* for tutorial information.

Conventions

This section describes the conventions that are used in this manual. By becoming familiar with these conventions, you will find it easier to gather information from this and other volumes in the documentation set.

The following conventions are covered:

- Typographical conventions
- Icon conventions
- Syntax conventions
- Sample-code conventions
- Terminology conventions

Typographical Conventions

This manual uses a standard set of conventions to emphasize words, present examples, describe statement syntax, and so forth. The following typographical conventions are used throughout this manual.

Convention	Meaning
italics	Within text, emphasized words are printed in italics. Within syntax diagrams, values that you are to specify are printed in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, table names, column names, menu items, command names, and other similar terms are printed in boldface.

(1 of 2)

Product and Platform Icons

Product and platform icons identify paragraphs that describe product-specific or platform-specific information. The following table describes the product and platform icons that are used in this manual.

Icon	Description
SE	Identifies information that is valid only for INFORMIX-SE.
D/B	Identifies information that is valid only for DB-Access.
ESQL	Identifies information that is valid only for SQL statements in INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL.
E/C	Identifies information that is valid only for INFORMIX-ESQL/C.
E/CO	Identifies information that is valid only for INFORMIX-ESQL/COBOL.
SPL	Identifies information that is valid only if you are using Informix Stored Procedure Language ( ).
OP	Identifies information that is valid only for INFORMIX-OnLine/Optical.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the product- or platform-specific information.

Compliance Icons

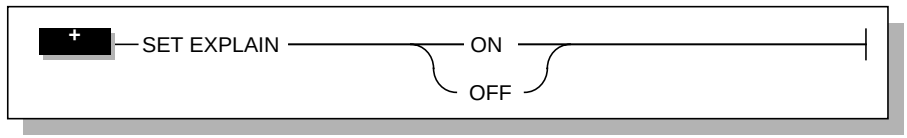
Compliance icons indicate paragraphs that provide guidelines for complying with a standard.

Icon	Description
ANSI	Identifies information that is valid only if your database is ANSI-compliant.
GLS	Identifies information that is valid only if your database or application uses a nondefault GLS locale.
X/O	Indicates that the functionality described in the text conforms to X/Open specifications for dynamic SQL. This functionality is available when you compile your SQL API with the **-xopen** flag.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the compliance information.

Syntax Conventions

This section describes conventions for syntax diagrams. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given statement, command line, or other specification, as in the following diagram of the SET EXPLAIN statement.



Each syntax diagram begins at the upper left corner and ends at the upper right corner with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement. (For a few diagrams, notes in the text identify path segments that are mutually exclusive.)

Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. Except for separators in loops, which the path approaches counterclockwise from the right, the path always approaches elements from the left and continues to the right. Unless otherwise noted, at least one blank character separates syntax elements.

Elements That Can Appear on the Path

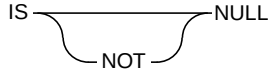
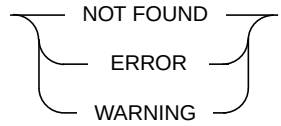
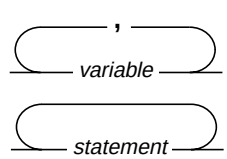
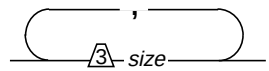
You might encounter one or more of the following elements on a path.

Element	Description
KEYWORD	A word in UPPERCASE characters is a keyword. You must spell the word exactly as shown; however, you can use either uppercase or lowercase characters.
(. , ; @ + * - /)	Punctuation and other non-alphanumeric characters are literal symbols that you must enter exactly as shown.
' '	Single quotes are literal symbols that you must enter as shown.
variable	A word in *italics* represents a value that you must supply. A table immediately following the diagram explains the value.
ADD Clause p. 1-14 ADD Clause	A term shown in a rectangle represents a subdiagram on the same page (if no page number is supplied) or a specified page, as if the subdiagram were spliced into the diagram at this point. The same subdiagram can be represented by rectangles of different shapes, as in these symbols for the ADD Clause subdiagram.
OL	An icon is a warning that this path is valid only for some products, or only under certain conditions. Characters on the icons indicate what products or conditions support the path. These icons might appear in a syntax diagram: OL This path is valid only for INFORMIX-OnLine Dynamic Server.

(1 of 3)

Element	Description	
	SE	This path is valid only for INFORMIX-SE.
	D/B	This path is valid only for DB-Access.
	ESQL	This path is valid only for SQL statements in INFORMIX-ESQL/C and INFORMIX-ESQL/COBOL.
	E/C	This path is valid only for INFORMIX-ESQL/C.
	E/CO	This path is valid only for INFORMIX-ESQL/COBOL.
	SPL	This path is valid only if you are using Informix Stored Procedure Language (SPL).
	OP	This path is valid only for INFORMIX-OnLine/Optical.
	+	This path is an Informix extension to ANSI SQL-92 entry-level standard SQL. If you initiate Informix extension checking and include this syntax branch, you receive a warning. If you have set the **DBANSIWARN** environment variable at compile time, or have used the **-ansi** compile flag, you receive warnings at compile time. If you have **DBANSIWARN** set at runtime, or if you compiled with the **-ansi** flag, warning flags are set in the **sqlwarn** structure.
	GLS	This path is valid only if your database or application uses a nondefault GLS locale.
	A shaded option is the default. If you do not specify any of the available options, this option is in effect by default.	
	Syntax that is enclosed between a pair of arrows is a subdiagram.	

(2 of 3)

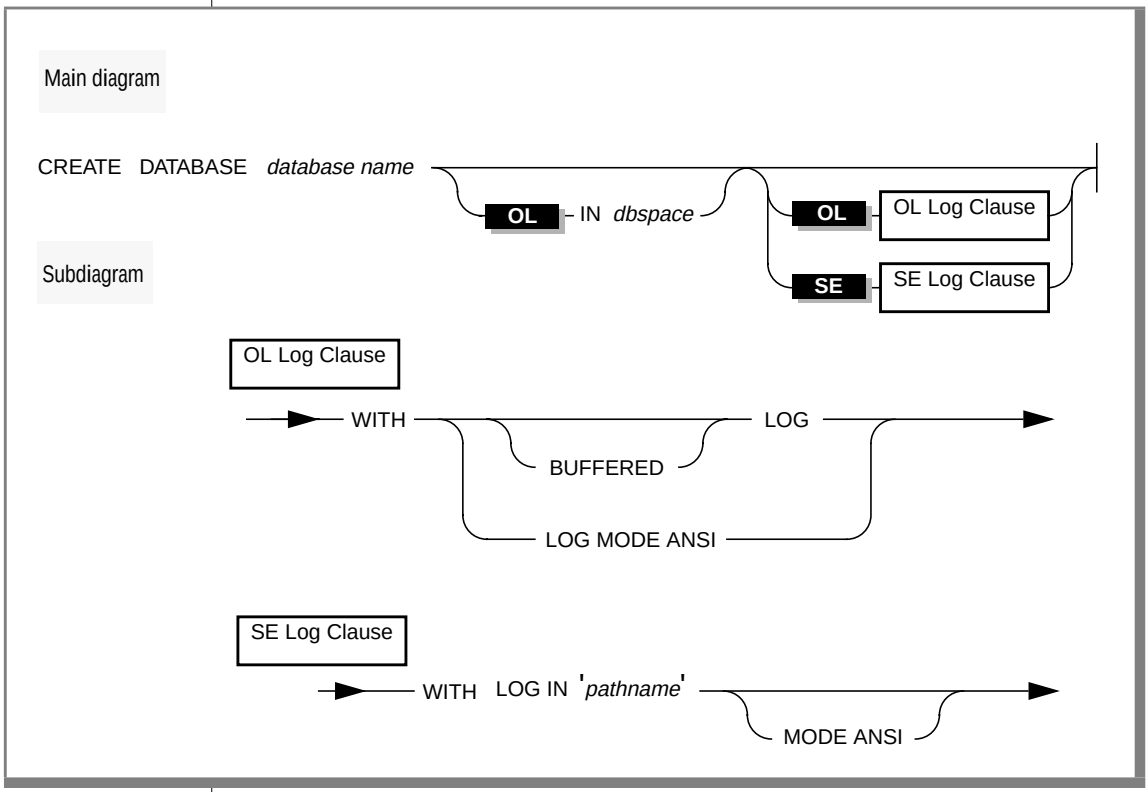
Element	Description
	The vertical line is a terminator. This symbol only appears at the right, indicating that the syntax diagram is complete.
	A branch below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)
	A set of multiple branches indicates that a choice among more than two different paths is available.
	A loop indicates a path that you can repeat. Punctuation along the top of the loop indicates the separator symbol for list items, as in this example. If no symbol appears, a blank space is the separator,
	A gate (3) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here you can specify *size* no more than three times within this statement segment.

(3 of 3)

How to Read a Syntax Diagram

Figure 1 shows a syntax diagram that uses most of the path elements that are listed in the previous table.

Figure 1
Elements of a Syntax Diagram



To use this diagram to construct a statement, start at the top left with the keywords CREATE DATABASE. Then follow the diagram to the right, proceeding through the options that you want.

To read the example syntax diagram

1. You must type the words CREATE DATABASE.
2. You must supply a *database name*.
3. You can stop, taking the direct route to the terminator, or you can take one or more of the optional paths.
4. If desired, you can designate a dbspace by typing the word IN and a dbspace name.
5. If desired, you can specify logging. Here, you are constrained by the database server with which you are working.
   - If you are using INFORMIX-OnLine Dynamic Server, go to the subdiagram named *OL Log Clause*. Follow the subdiagram by typing the keyword WITH, then choosing and typing either LOG, BUFFERED LOG, or LOG MODE ANSI. Then, follow the arrow back to the main diagram.
   - If you are using INFORMIX-SE, go to the subdiagram named *SE Log Clause*. Follow the subdiagram by typing the keywords WITH LOG IN, typing a quote, supplying a pathname, and closing the quotes. You can then choose the MODE ANSI option below the line or continue to follow the line across.
6. Once you are back at the main diagram, you come to the terminator. Your CREATE DATABASE statement is complete.

Sample-Code Conventions

Examples of SQL code occur throughout this manual. Except where noted, the code is not specific to any single Informix application development tool. If only SQL statements are listed in the example, they are not delimited by semicolons. To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using the Query-language option of DB-Access, you must delimit multiple statements with semicolons. If you are using an SQL API, you must use EXEC SQL and a semicolon (or other appropriate delimiters) at the start and end of each statement, respectively.

For instance, you might see the code in the following example:

```
CONNECT TO stores7
.
.
.
DELETE FROM customer
    WHERE customer_num = 121
.
.
.
COMMIT WORK
DISCONNECT CURRENT
```

Dots in the example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the manual for your product.

Terminology Conventions

This manual uses a standard set of conventions for terms and abbreviations.

Definitions of Terms

For definitions of terms used in this manual and in the other manuals of the SQL series, see the Glossary in the *Informix Guide to SQL: Reference*.

Abbreviations of Product Names

The following abbreviations for product names and features appear in this manual.

Abbreviation	Complete Name
GLS	Global Language Support
OnLine	INFORMIX-OnLine Dynamic Server
SE	INFORMIX-SE
SQL	Structured Query Language

Additional Documentation

This section describes the following pieces of the documentation set:

- Printed documentation
- On-line documentation
- Related reading

Printed Documentation

In addition to this manual, the following printed manuals are included in the SQL manual series:

- A companion volume to this manual, the *Informix Guide to SQL: Reference*, provides reference information on the types of Informix databases you can create, the data types supported in Informix products, system catalog tables associated with the database, and environment variables you can set to use your Informix products properly. This manual also provides a detailed description of the **stores7** demonstration database and contains a glossary.
- An additional companion volume to this manual, the *Informix Guide to SQL: Tutorial*provides a tutorial on SQL as it is implemented by Informix products. It describes the fundamental ideas and terminology that are used when planning, using, and implementing a relational database.
- The *SQL Quick Syntax Guide* contains syntax diagrams for all statements and segments described in this manual.

The following related Informix documents complement the information in this manual set:

- *Getting Started with Informix Database Server Products* provides an orientation to the Informix client / server environment and describes the manuals for Informix products. If you are a new user of Informix products, it is helpful to read this manual before you read any of the manuals in the SQL manual series.
- The *Guide to GLS Functionality* explains the impact of the GLS feature on Informix products. This manual includes a chapter on SQL features and a chapter on GLS environment variables.
- You, or whoever installs your Informix products, should refer to the *UNIX Products Installation guide* for your particular release to ensure that your Informix product is properly set up before you begin to work with it. A matrix that depicts possible client / server configurations is included in the *UNIX Products Installation guide*.
- Depending on the database server you are using, you or your system administrator need either the*INFORMIX-SE Administrator's Guide* or the *INFORMIX-OnLine Dynamic Server Administrator's Guide*. The *DB-Access User Manual* describes how to invoke the DB-Access utility to access, modify, and retrieve information from Informix database servers.
- When errors occur, you can look them up by number and learn their cause and solution in the *Informix Error Messages* manual. If you prefer, you can look up the error messages in the on-line message file described in the section "Error Message Files" later in this Introduction and in the Introduction to the *Informix Error Messages* manual.

On-Line Documentation

The following online files supplement this document:

- On-line error messages
- Release notes, documentation notes, and machine notes

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. To read the error messages in the ASCII file, Informix provides scripts that let you display error messages on the screen (**finderr**) or print formatted error messages (**rofferr**). See the Introduction to the *Informix Error Messages* manual for a detailed description of these scripts.

The optional Informix Messages and Corrections product provides PostScript files that contain the error messages and their corrective actions. If you have installed this product, you can print the PostScript files on a PostScript printer. The PostScript error messages are distributed in a number of files of the format **errmsg1.ps**, **errmsg2.ps**, and so on. These files are located in the **$INFORMIXDIR/msg** directory.

Release Notes, Documentation Notes, Machine Notes

In addition to the Informix set of manuals, the following on-line files, located in the **$INFORMIXDIR/release/en_us/0333** directory, supplement the information in this manual.

On-Line File	Purpose
Documentation Notes	Describes features that are not covered in the manual or that have been modified since publication. The file that contains the documentation notes for this product is called **SQLSDOC_7.2.**
Release Notes	Describes feature differences from earlier versions of Informix products and how these differences might affect current products. The file that contains the release notes for Version 7..2 of Informix database server products is called **SERVERS_7.2.**
Machine Notes	Describes any special actions required to configure and use Informix products on your computer. Machine notes are named for the product described. For example, the machine notes file for INFORMIX-OnLine Dynamic Server is **ONLINE_7.2.**

Please examine these files because they contain vital information about application and performance issues.

Related Reading

For additional technical information on database management, consult the following books. The first book is an introductory text for readers who are new to database management, while the second book is a more complex technical work for SQL programmers and database administrators:

- *Database: A Primer* by C. J. Date (Addison-Wesley Publishing, 1983)
- *An Introduction to Database Systems* by C. J. Date (Addison-Wesley Publishing, 1994).

To learn more about the SQL language, consider the following books:

- *A Guide to the SQL Standard* by C. J. Date with H. Darwen (Addison-Wesley Publishing, 1993)
- *Understanding the New SQL: A Complete Guide* by J. Melton and A. Simon (Morgan Kaufmann Publishers, 1993)
- *Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

This manual assumes that you are familiar with your computer operating system. If you have limited UNIX system experience, consult your operating system manual or a good introductory text before you read this manual. The following texts provide a good introduction to UNIX systems:

- *Introducing the UNIX System* by H. McGilton and R. Morgan (McGraw-Hill Book Company, 1983)
- *Learning the UNIX Operating System* by G. Todino, J. Strang, and J. Peek (O'Reilly & Associates, 1993)
- *A Practical Guide to the UNIX System* by M. Sobell (Benjamin / Cummings Publishing, 1989)
- *UNIX for People* by P. Birns, P. Brown, and J. Muster (Prentice-Hall, 1985)
- *UNIX System V: A Practical Guide* by M. Sobell (Benjamin / Cummings Publishing, 1995)

Compliance with Industry Standards

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992 on INFORMIX-OnLine Dynamic Server. In addition, many features of OnLine comply with the SQL-92 Intermediate and Full Level and X/Open CAE (common applications environment) standards.

Informix SQL-based products are compliant with ANSI SQL-92 Entry Level (published as ANSI X3.135-1992) on INFORMIX-SE with the following exceptions:

- Effective checking of constraints
- Serializable transactions

Informix Welcomes Your Comments

Please let us know what you like or dislike about our manuals. To help us with future versions of our manuals, please tell us about any corrections or clarifications that you would find useful. Write to us at the following address:

Informix Software, Inc.
SCT Technical Publications Department
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send electronic mail, our address is:

doc@informix.com

Or send a facsimile to the Informix Technical Publications Department at:

415-926-6571

Please include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

We appreciate your feedback.

Chapter

1

SQL Statements

How to Enter SQL Statements 1-6

How to Enter SQL Comments 1-9

Categories of SQL Statements 1-12

ANSI Compliance and Extensions 1-16

Statements 1-18
- ALLOCATE DESCRIPTOR 1-19
- ALTER FRAGMENT 1-22
- ALTER INDEX 1-43
- ALTER TABLE 1-46
- BEGIN WORK 1-77
- CHECK TABLE 1-79
- CLOSE 1-81
- CLOSE DATABASE 1-85
- COMMIT WORK 1-87
- CONNECT 1-89
- CREATE AUDIT 1-102
- CREATE DATABASE 1-104
- CREATE INDEX 1-109
- CREATE PROCEDURE 1-134
- CREATE PROCEDURE FROM 1-144
- CREATE ROLE 1-145
- CREATE SCHEMA 1-147
- CREATE SYNONYM 1-150
- CREATE TABLE 1-154
- CREATE TRIGGER 1-192
- CREATE VIEW 1-224

DATABASE 1-229
DEALLOCATE DESCRIPTOR 1-232
DECLARE 1-234
DELETE 1-252
DESCRIBE 1-255
DISCONNECT 1-261
DROP AUDIT 1-265
DROP DATABASE 1-266
DROP INDEX 1-268
DROP PROCEDURE 1-270
DROP ROLE 1-271
DROP SYNONYM 1-272
DROP TABLE 1-274
DROP TRIGGER 1-277
DROP VIEW 1-279
EXECUTE 1-281
EXECUTE IMMEDIATE 1-290
EXECUTE PROCEDURE 1-293
FETCH 1-296
FLUSH 1-308
FREE 1-311
GET DESCRIPTOR 1-314
GET DIAGNOSTICS 1-321
GRANT 1-340
GRANT FRAGMENT 1-356
INFO 1-365
INSERT 1-370
LOAD 1-380
LOCK TABLE 1-387
OPEN 1-390
OUTPUT 1-400
PREPARE 1-402
PUT 1-416
RECOVER TABLE 1-425

RENAME COLUMN 1-428
RENAME DATABASE 1-431
RENAME TABLE. 1-432
REPAIR TABLE 1-435
REVOKE 1-437
REVOKE FRAGMENT 1-450
ROLLBACK WORK 1-455
ROLLFORWARD DATABASE 1-457
SELECT 1-459
SET 1-501
SET CONNECTION 1-527
SET DATASKIP 1-534
SET DEBUG FILE TO 1-537
SET DESCRIPTOR 1-540
SET EXPLAIN 1-548
SET ISOLATION 1-556
SET LOCK MODE 1-561
SET LOG 1-564
SET OPTIMIZATION 1-566
SET PDQPRIORITY 1-568
SET ROLE 1-571
SET SESSION AUTHORIZATION 1-573
SET TRANSACTION 1-575
START DATABASE 1-581
START VIOLATIONS TABLE 1-584
STOP VIOLATIONS TABLE 1-603
UNLOAD 1-605
UNLOCK TABLE. 1-610
UPDATE. 1-612
UPDATE STATISTICS 1-623
WHENEVER 1-632

Segments 1-640
Condition 1-643
Constraint Name 1-658
Database Name 1-660
Data Type 1-664
DATETIME Field Qualifier 1-669
Expression 1-671
Identifier 1-723
Index Name 1-741
INTERVAL Field Qualifier 1-743
Literal DATETIME 1-746
Literal INTERVAL 1-749
Literal Number 1-752
Procedure Name 1-754
Quoted String 1-757
Relational Operator 1-761
Synonym Name 1-766
Table Name 1-768
View Name 1-772

This chapter provides comprehensive reference information about SQL statements and the SQL segments that recur in SQL statements. It is organized into the following sections:

- "How to Enter SQL Statements" shows how to use the information in the statement descriptions to enter SQL statements correctly.
- "How to Enter SQL Comments" shows how to enter comments for your SQL statements in DB-Access command files, ESQL programs, and stored procedures.
- "Categories of SQL Statements" divides SQL statements into several functional categories and lists the statements within each category. Some examples of these categories are data definition statements, data manipulation statements, and data integrity statements.
- "ANSI Compliance and Extensions" explains how the SQL statements in this manual comply with the ANSI SQL standard. This section provides a list of ANSI-compliant statements, a list of ANSI-compliant statements with Informix extensions, and a list of statements that are Informix extensions to the ANSI standard.
- "Statements" gives comprehensive descriptions of SQL statements. The statements are listed in alphabetical order.
- "Segments" gives comprehensive descriptions of SQL segments. The segments are listed in alphabetical order. SQL segments are language elements, such as table names and expressions, that occur in many SQL statements. Instead of describing each segment in each statement where it occurs, this chapter provides a comprehensive stand-alone description of each segment. Whenever a segment occurs within the syntax diagram for an SQL statement, the diagram points to the stand-alone description of the segment for further information.

The following table summarizes the sections of this chapter.

Section	Starting Page	Scope
"How to Enter SQL Statements"	1-6	This section shows how to use the statement descriptions to enter SQL statements correctly.
"How to Enter SQL Comments"	1-9	This section shows how to enter comments for SQL statements.
"Categories of SQL Statements"	1-12	This section lists SQL statements by functional category.
"ANSI Compliance and Extensions"	1-16	This section lists SQL statements by degree of ANSI compliance.
"Statements"	1-18	This section gives reference descriptions of all SQL statements.
"Segments"	1-640	This section gives reference descriptions of all SQL segments.

How to Enter SQL Statements

The purpose of the statement descriptions in this chapter is to help you to enter SQL statements successfully and to understand the behavior of the statements. Each statement description includes the following information:

- A brief introduction that explains the purpose of the statement
- A syntax diagram that shows how to enter the statement correctly
- A syntax table that explains each input parameter in the syntax diagram
- Rules of usage, including examples that illustrate these rules

If a statement consists of multiple clauses, the statement description provides the same set of information for each clause.

Each statement description concludes with references to related information in this manual and other manuals.

The major aids for entering SQL statements successfully include:

- the combination of the syntax diagram and syntax table.
- the examples of syntax that appear in the rules of usage.
- the references to related information.

Using Syntax Diagrams and Syntax Tables

Before you try to use the syntax diagrams in this chapter, it is helpful to read the "Syntax Conventions" on page 19 of the Introduction. This section is the key to understanding the syntax diagrams in the statement descriptions.

The "Syntax Conventions" section explains the elements that can appear in a syntax diagram and the paths that connect the elements to each other. This section also includes a sample syntax diagram that illustrates the major elements of all syntax diagrams. The narrative that follows the sample diagram shows how to read the diagram in order to enter the statement successfully.

When a syntax diagram within a statement description includes input parameters, the syntax diagram is followed by a syntax table that shows how to enter the parameters without generating errors. Each syntax table includes the following columns:

- The **Elements** column lists the name of each parameter as it appears in the syntax diagram.
- The **Purpose** column briefly states the purpose of the parameter. If the parameter has a default value, it is listed in this column.
- The **Restrictions** column summarizes the restrictions on the parameter, such as acceptable ranges of values.
- The **Syntax** column points to the SQL segment that gives the detailed syntax for the parameter.

Using Examples

To understand the main syntax diagram and subdiagrams for a statement, study the examples of syntax that appear in the rules of usage for each statement. These examples have two purposes:

- To show how to accomplish particular tasks with the statement or its clauses
- To show how to use the syntax of the statement or its clauses in a concrete way

Tip: *An efficient way to understand a syntax diagram is to find an example of the syntax and compare it with the keywords and parameters in the syntax diagram. By mapping the concrete elements of the example to the abstract elements of the syntax diagram, you can understand the syntax diagram and use it more effectively.*

For an explanation of the conventions used in the examples in this manual, see "Sample-Code Conventions" on page 24 of the Introduction.

Using References

For help in understanding the concepts and terminology in the SQL statement description, check the "References" section at the end of the description.

The "References" section points to related information in this manual and other manuals that helps you to understand the statement in question. This section provides some or all of the following information:

- The names of related statements that might contain a fuller discussion of topics in this statement
- The titles of other manuals that provide extended discussions of topics in this statement
- The chapters in the *Informix Guide to SQL: Tutorial* that provide a task-oriented discussion of topics in this statement

Tip: *If you do not have extensive knowledge and experience with SQL, the "Informix Guide to SQL: Tutorial" gives you the basic SQL knowledge that you need to understand and use the statement descriptions in this manual.*

How to Enter SQL Comments

You can add comments to clarify the purpose or effect of particular SQL statements. Your comments can help you or others to understand the role of the statement within a program, stored procedure, or command file. The code examples in this manual sometimes include comments that clarify the role of an SQL statement within the code.

The following table shows the SQL comment symbols that you can enter in your code. A *Y* in a column signifies that you can use the symbol with the product or database type named in the column heading. An *N* in a column signifies that you cannot use the symbol with the product or database type that the column heading names.

Comment Symbol	SQL APIs	Stored Procedures (SPL)	DB-Access	ANSI-Compliant Databases	Databases That Are Not ANSI Compliant	Description
double dash (--)	Y	Y	Y	Y	Y	The double dash precedes the comment. The double dash can comment only a single line. If you want to use the double dash to comment more than one line, you must put the double dash at the beginning of each comment line.
curly brackets ({})	N	Y	Y	Y	Y	Curly brackets enclose the comment. The { precedes the comment, and the } follows the comment. You can use curly brackets for single-line comments or for multiple-line comments.

If the product that you are using supports both comment symbols, your choice of a comment symbol depends on your requirements for ANSI compliance:

- The double dash (--) complies with the ANSI SQL standard.
- Curly brackets ({}) are an Informix extension to the standard.

If ANSI compliance is not an issue, your choice of comment symbols is a matter of personal preference.

D/B

You can use either comment symbol when you enter SQL statements with the SQL editor and when you create SQL command files with the SQL editor or a system editor. An SQL command file is an operating-system file that contains one or more SQL statements. Command files are also known as command scripts. For more information about command files, see the discussion of command scripts in the *Informix Guide to SQL: Tutorial*. For information on creating and modifying command files with the SQL editor or a system editor in DB-Access, see the *DB-Access User Manual*. ♦

SPL

You can use either comment symbol in any line of a stored procedure. See "Adding Comments to the Procedure" on page 1-138 for further information. Also see the discussion of commenting and documenting a procedure in the *Informix Guide to SQL: Tutorial*. ♦

ESQL

You can use the double dash (--) to comment SQL statements in your SQL API. See the manual for your SQL API for further information on the use of SQL comment symbols and language-specific comment symbols in application programs. ♦

Examples of SQL Comment Symbols

Some simple examples can help to illustrate the different ways of using the SQL comment symbols.

Examples of the Double-Dash Symbol

The following example shows the use of the double dash (--) to comment an SQL statement. In this example, the comment appears on the same line as the statement.

```
SELECT * FROM customer -- Selects all columns and rows
```

In the following example, the user enters the same SQL statement and the same comment as in the preceding example, but the user places the comment on a line by itself:

```
SELECT * FROM customer
    -- Selects all columns and rows
```

In the following example, the user enters the same SQL statement as in the preceding example but now enters a multiple-line comment:

```
SELECT * FROM customer
    -- Selects all columns and rows
    -- from the customer table
```

Examples of the Curly-Brackets Symbols

D/B
SPL

The following example shows the use of curly brackets ({}) to comment an SQL statement. In this example, the comment appears on the same line as the statement.

```
SELECT * FROM customer {Selects all columns and rows}
```

In the following example, the user enters the same SQL statement and the same comment as in the preceding example but places the comment on a line by itself:

```
SELECT * FROM customer
    {Selects all columns and rows}
```

In the following example, the user enters the same SQL statement as in the preceding example but enters a multiple-line comment:

```
SELECT * FROM customer
    {Selects all columns and rows
     from the customer table}
```

♦

Non-ASCII Characters in SQL Comments

GLS

You can enter non-ASCII characters (including multibyte characters) in SQL comments if your locale supports a code set with the non-ASCII characters. See the *Guide to GLS Functionality* for further information on the GLS aspects of SQL comments. ♦

Categories of SQL Statements

SQL statements are divided into the following categories:

- Data definition statements
- Data manipulation statements
- Cursor manipulation statements
- Dynamic management statements
- Data access statements
- Data integrity statements
- Query optimization information statements
- Stored procedure statements
- Auxiliary statements
- Client / server connection statements
- Optical statements

The specific statements for each category are as follows.

Data Definition Statements

ALTER FRAGMENT
ALTER INDEX
ALTER TABLE
CLOSE DATABASE
CREATE DATABASE
CREATE INDEX
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE ROLE
CREATE SCHEMA
CREATE SYNONYM
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DATABASE
DROP DATABASE

DROP INDEX
DROP PROCEDURE
DROP ROLE
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP VIEW
RENAME COLUMN
RENAME DATABASE
RENAME TABLE

Data Manipulation Statements

DELETE
INSERT
LOAD
SELECT
UNLOAD
UPDATE

Cursor Manipulation Statements

CLOSE
DECLARE
FETCH
FLUSH
FREE
OPEN
PUT

Dynamic Management Statements

ALLOCATE DESCRIPTOR
DEALLOCATE DESCRIPTOR
DESCRIBE
EXECUTE
EXECUTE IMMEDIATE
FREE
GET DESCRIPTOR
PREPARE
SET DESCRIPTOR

Data Access Statements

GRANT
GRANT FRAGMENT
LOCK TABLE
REVOKE
REVOKE FRAGMENT
SET ISOLATION
SET LOCK MODE
SET ROLE
SET SESSION AUTHORIZATION
SET TRANSACTION
UNLOCK TABLE

Data Integrity Statements

BEGIN WORK
CHECK TABLE
COMMIT WORK
CREATE AUDIT
DROP AUDIT
RECOVER TABLE
REPAIR TABLE
ROLLBACK WORK
ROLLFORWARD DATABASE
SET
SET LOG
START DATABASE
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE

Query Optimization Information Statements

SET EXPLAIN
SET OPTIMIZATION
SET PDQPRIORITY
UPDATE STATISTICS

Stored Procedure Statements

EXECUTE PROCEDURE
SET DEBUG FILE TO

Auxiliary Statements

INFO
OUTPUT
GET DIAGNOSTICS
WHENEVER

Client/Server Connection Statements

CONNECT
DISCONNECT
SET CONNECTION

INFORMIX-OnLine/Optical Statements

ALTER OPTICAL CLUSTER
CREATE OPTICAL CLUSTER
DROP OPTICAL CLUSTER
RELEASE
RESERVE
SET MOUNTING TIMEOUT

Important: *INFORMIX-OnLine/Optical statements are described in the "INFORMIX-OnLine/Optical User Manual."*

ANSI Compliance and Extensions

The following lists show statements that are compliant with the ANSI SQL-92 standard at the entry level, statements that are ANSI compliant but include Informix extensions, and statements that are Informix extensions to the ANSI standard.

ANSI-Compliant Statements

CLOSE
COMMIT WORK
ROLLBACK WORK
SET SESSION AUTHORIZATION
SET TRANSACTION

ANSI-Compliant Statements with Informix Extensions

CREATE SCHEMA AUTHORIZATION
CREATE TABLE
CREATE VIEW
DECLARE
DELETE
EXECUTE
FETCH
GRANT
INSERT
OPEN
SELECT
SET CONNECTION
UPDATE
WHENEVER

Statements That Are Extensions to the ANSI Standard

ALLOCATE DESCRIPTOR
ALTER FRAGMENT
ALTER INDEX
ALTER OPTICAL CLUSTER
ALTER TABLE
BEGIN WORK
CHECK TABLE

CLOSE DATABASE
CONNECT
CREATE AUDIT
CREATE DATABASE
CREATE INDEX
CREATE OPTICAL CLUSTER
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE ROLE
CREATE SYNONYM
CREATE TRIGGER
DATABASE
DEALLOCATE DESCRIPTOR
DESCRIBE
DISCONNECT
DROP AUDIT
DROP DATABASE
DROP INDEX
DROP OPTICAL CLUSTER
DROP PROCEDURE
DROP ROLE
DROP SYNONYM
DROP TABLE
DROP TRIGGER
DROP VIEW
EXECUTE IMMEDIATE
EXECUTE PROCEDURE
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
GRANT FRAGMENT
INFO
LOAD
LOCK TABLE
OUTPUT
PREPARE
PUT
RECOVER TABLE
RELEASE
RENAME COLUMN
RENAME DATABASE
RENAME TABLE
RESERVE

REVOKE
REVOKE FRAGMENT
ROLLFORWARD DATABASE
SET
SET DATASKIP
SET DEBUG FILE TO
SET DESCRIPTOR
SET EXPLAIN
SET ISOLATION
SET LOCK MODE
SET LOG
SET OPTIMIZATION
SET PDQPRIORITY
SET ROLE
SET TRANSACTION
START DATABASE
START VIOLATIONS TABLE
STOP VIOLATIONS TABLE
UNLOAD
UNLOCK TABLE
UPDATE STATISTICS

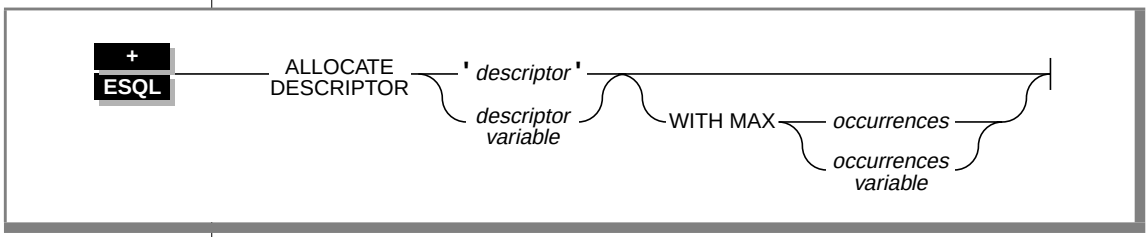
Statements

This section gives comprehensive reference descriptions of SQL statements. The statement descriptions appear in alphabetical order. For an explanation of the structure of statement descriptions, see "How to Enter SQL Statements" on page 1-6.

ALLOCATE DESCRIPTOR

Use the ALLOCATE DESCRIPTOR statement to allocate memory for a system-descriptor area. A *descriptor* parameter or a *descriptor variable* parameter identifies the system-descriptor area. This statement creates a place in memory to hold information that a DESCRIBE statement obtains or to hold information about the WHERE clause of a statement.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	Use single quotes. String must represent the name of an unallocated system-descriptor area.	Quoted String, p. 1-757
descriptor variable	Host-variable name that identifies a system-descriptor area	Variable must contain the name of an unallocated system-descriptor area.	Name must conform to language-specific rules for variable names.
occurrences	The number of item descriptors in the system-descriptor area	Value must be unsigned INTEGER. Default value is 100.	Literal Number, p. 1-752
occurrences variable	Host variable that contains the number of *occurrences*	Data type must be INTEGER or SMALLINT.	Name must conform to language-specific rules for variable names.

Usage

The ALLOCATE DESCRIPTOR statement creates a system-descriptor area. The *descriptor* parameter or the *descriptor variable* parameter identifies this area.

A system-descriptor area contains one or more fields called item descriptors. Each item descriptor holds a data value that the database server can receive or send. The item descriptors also contain information about the data such as type, length, scale, precision, and nullability.

Either the *occurrences* parameter or the *occurrences variable* parameter specifies the number of item descriptors that you want in the system descriptor area.

Initially, all fields in the item-descriptor area are undefined. The COUNT field is set to the number of occurrences that you specified in the *occurrences* parameter or the *occurrences variable* parameter. The TYPE field, the LENGTH field, and other fields in the item descriptor are set when a DESCRIBE statement is executed using the system descriptor. The DESCRIBE statement also allocates memory for the DATA field in each item descriptor, based on the TYPE and LENGTH information. You can also use item descriptors with stored procedures. A DESCRIBE statement obtains information for the stored procedures. See Chapter 2, "SPL Statements," for more information about stored procedures.

If the name that you assign to a system-descriptor area matches the name of an existing system-descriptor area, the database server returns an error. If you free the descriptor with the DEALLOCATE DESCRIPTOR statement, you can reuse the descriptor.

The WITH MAX Clause

You can use the optional WITH MAX *occurrences* clause to indicate the number of value descriptors you need. This number must be greater than zero. When you do not specify the WITH MAX clause, the database server uses a default value of 100 for the *occurrences* parameter.

The following examples show the ALLOCATE DESCRIPTOR statement for two SQL APIs. Each example includes the WITH MAX clause.

In each pair of examples, the first line uses an embedded variable name to identify the system-descriptor area, and the second line uses a quoted string to identify the system-descriptor area. In addition, in each pair of examples, the first line uses an embedded variable name to specify the desired number of item descriptors, and the second line uses an unsigned integer to specify the desired number of item descriptors.

INFORMIX-ESQL/C

```
EXEC SQL allocate descriptor :descname with max :occ;

EXEC SQL allocate descriptor 'desc1' with max 3;
```

INFORMIX-ESQL/COBOL

```
EXEC SQL ALLOCATE DESCRIPTOR :DESCNAME WITH MAX :OCC END-EXEC.

EXEC SQL ALLOCATE DESCRIPTOR 'DESC1' WITH MAX 3 END-EXEC.
```

References

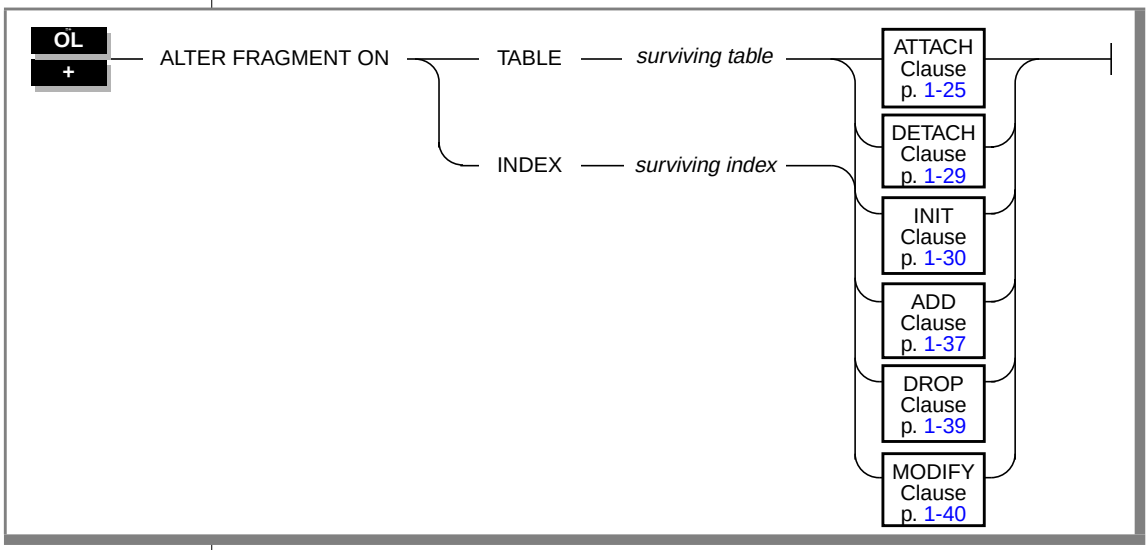
See the DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of system-descriptor areas in Chapter 5.

ALTER FRAGMENT

Use the ALTER FRAGMENT statement to alter an existing table or index fragmentation strategy dynamically as well as to create fragments initially.

Syntax



Element	Purpose	Restrictions	Syntax
surviving index	The index on which you execute the ALTER FRAGMENT statement	The index must exist at the time you execute the statement.	Index Name, p. 1-741
surviving table	The table on which you execute the ALTER FRAGMENT statement	The table must exist at the time you execute the statement.	Table Name, p. 1-768

Usage

You can alter the fragmentation strategy of an existing table or index, or you can create a new fragmentation strategy for nonfragmented tables. Use the ALTER FRAGMENT statement to tune your fragmentation strategy. The clauses of the ALTER FRAGMENT statement let you perform the following tasks.

Clause	Purpose
ATTACH	Combines tables that contain identical table structures into a single fragmented table.
DETACH	Detaches a table fragment from a fragmentation strategy and place it in a new table.
INIT	Defines and initializes a new fragmentation strategy on a nonfragmented table or converts an existing fragmentation strategy on a fragmented table. You can also use this clause to change the order of evaluation of fragment expressions.
ADD	Adds an additional fragment to an existing fragmentation list.
DROP	Drops an existing fragment from a fragmentation list.
MODIFY	Changes an existing fragmentation expression.

You must have the Alter or the DBA privilege to change the fragmentation strategy of a table. You must have the Index or the DBA privilege to alter the fragmentation strategy of an index.

The ALTER FRAGMENT statement applies only to table or index fragments that are located at the current site. No remote information is accessed or updated.

INIT and ATTACH are the only operations you can perform for tables and indexes that are not already fragmented.

You cannot use the ALTER FRAGMENT statement on a temporary table or a view.

How Is the ALTER FRAGMENT Statement Executed?

If your database uses logging, the ALTER FRAGMENT statement is executed within a single transaction. When the fragmentation strategy uses large numbers of records, you might run out of log space or disk space (INFORMIX-OnLine Dynamic Server requires extra disk space for the operation; it later frees the disk space).

Making More Space

When you run out of log space or disk space, try one of the following procedures to make more space available:

- Turn off logging and turn it back on again at the end of the operation. This procedure indirectly requires a backup of the root dbspace.

  For more information about the **ontape** utility to start and stop logging, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*. You can also use the ON-Archive utility to back up the root dbspace. For more information about ON-Archive, see the *INFORMIX-OnLine Dynamic Server Archive and Backup Guide*.
- Split the operations into multiple ALTER FRAGMENT statements, moving a smaller portion of records at each time.

See the *INFORMIX-OnLine Dynamic Server Administrator's Guide* for information about log-space requirements and disk-space requirements. That guide also contains detailed instructions about how to turn off logging.

Determining the Number of Rows in the Fragment

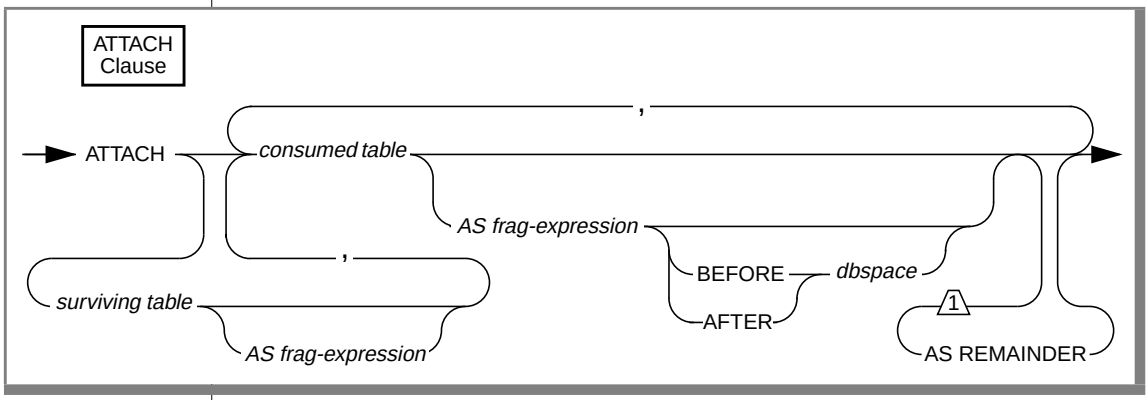
You can place as many rows into a fragment as the available space in the dbspace allows. To find out how many rows are in a fragment, perform these simple steps:

1. Run the UPDATE STATISTICS statement on the table. This step fills the **sysfragments** system catalog table with the current table information.
2. Query the **sysfragments** system catalog table to examine the **npused** and **nrows** fields. The **npused** field gives you the number of data pages used in the fragment, and the **nrows** field gives you the number of rows in the fragment.

The ATTACH Clause

Important: *Use the CREATE TABLE statement or the ALTER FRAGMENT INIT statement to create fragmented tables.*

Use the ATTACH clause to combine tables that contain identical table structures into a fragmentation strategy. Transforming tables with identical table structures into fragments in a single table allows OnLine to manage the fragmentation instead of the application managing the fragmentation. The distribution scheme can be either round-robin or expression based.



Element	Purpose	Restrictions	Syntax
consumed table	A nonfragmented table on which you execute the ATTACH clause	The table must exist at the time you execute the statement. No serial columns, referential constraints, primary-key constraints, or unique constraints are allowed in the table. The table can have check constraints and not null constraints, but these constraints are dropped after the ATTACH clause is executed.	Table Name, p. 1-768
dbspace	The dbspace name that specifies where the consumed table expression occurs in the fragmentation list	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-723

(1 of 2)

Any tables that you attach must have been created previously in separate dbspaces. You cannot attach the same table more than once. You cannot attach a fragmented table to another fragmented table.

You must be the DBA or the owner of the tables that are involved to use the ATTACH clause.

You cannot attach temporary tables.

After the tables are attached, the consumed table that is specified on the ATTACH clause no longer exists. The records that were in the consumed table must be referenced through the surviving table that is specified in the ALTER FRAGMENT ON TABLE statement.

Each table that is described in the ATTACH clause must be identical in structure; that is, all column definitions must match. The number, names, data types, and relative position of the columns must be identical. However, you cannot attach tables that contain serial columns. In addition, indexes and triggers on the surviving table survive the ATTACH, but indexes and triggers on the consumed table are dropped. Triggers are not activated with the ATTACH clause.

Combining Identically Structured Nonfragmented Tables

To make a single, fragmented table from two or more nonfragmented tables, the ATTACH clause must contain the surviving table as the first element of the *attach list*. The attach list is the list of tables in the ATTACH clause. For example, if you attach the tables **cur_acct** and **new_acct,** which were previously created in separate dbspaces, the surviving table **cur_acct** must be the first element in the attach list. The following statement illustrates this rule:

```
ALTER FRAGMENT ON TABLE cur_acct ATTACH cur_acct, new_acct
```

If you want a new rowid column on the single fragmented table, attach all tables first and then add the rowid with the ALTER TABLE statement.

Attaching a Nonfragmented Table to a Fragmented Table

To attach a nonfragmented table to an already fragmented table, the nonfragmented table must have been created in a separate dbspace and must have the same table structure as the fragmented table. The following example shows how to attach a nonfragmented table, **old_acct**, which was previously created in **dbsp3**, to a fragmented table, **cur_acct**:

```
ALTER FRAGMENT ON TABLE cur_acct ATTACH old_acct
```

The BEFORE and AFTER Clauses

The BEFORE and AFTER clauses allow you to place a new fragment in a dbspace either before or after an existing dbspace. Use the BEFORE and AFTER clauses only when the distribution scheme is expression based (not round-robin). Attaching a new fragment without an explicit BEFORE or AFTER clause places the added fragment at the end of the fragmentation list. You cannot attach a new fragment after the remainder fragment.

Using ATTACH to Fragment Tables: Round-Robin

The following example combines nonfragmented tables **pen_types** and **pen_makers** into a single, fragmented table, **pen_types**. Table **pen_types** resides in dbspace **dbsp1,** and table **pen_makers** resides in dbspace **dbsp2**. Table structures are identical in each table.

```
ALTER FRAGMENT ON TABLE pen_types
    ATTACH pen_types, pen_makers
```

After you execute the ATTACH clause, OnLine fragments the table **pen_types** round-robin into two dbspaces: the dbspace that contained **pen_types** and the dbspace that contained **pen_makers**. Table **pen_makers** is consumed, and no longer exists; all rows that were in table **pen_makers** are now in table **pen_types**.

Using ATTACH to Fragment Tables: Fragment Expression

Consider the following example that combines tables **cur_acct** and **new_acct** and uses an expression-based distribution scheme. Table **cur_acct** was originally created as a fragmented table and has fragments in dbspaces **dbsp1** and **dbsp2**. The first statement of the example shows that table **cur_acct** was created with an expression-based distribution scheme. The second statement of the example creates table **new_acct** in **dbsp3** without a fragmentation strategy. The third statement combines the tables **cur_acct** and **new_acct**. Table structures (columns) are identical in each table.

```
CREATE TABLE cur_acct (a int) FRAGMENT BY EXPRESSION
    a < 5 in dbsp1,
    a >=5 and a < 10 in dbsp2;

CREATE TABLE new_acct (a int) IN dbsp3;

ALTER FRAGMENT ON TABLE cur_acct ATTACH new_acct AS a>=10;
```

When you examine the **sysfragments** system catalog table after you have altered the fragment, you see that table **cur_acct** is fragmented by expression into three dbspaces. For additional information about the **sysfragments** system catalog table, see Chapter 2 of the *Informix Guide to SQL: Reference*.

In addition to simple range rules, you can use the ATTACH clause to fragment by expression with hash or arbitrary rules. For a discussion of all types of expressions in an expression-based distribution scheme, see "The FRAGMENT BY Clause for Tables" on page 1-35.

Warning: *When you specify a date value in a fragment expression, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

What Happens to Blob Columns?

Each blob column in every table that is named in the ATTACH clause must have the same storage type, either blobspace or tblspace. If the blob column is a blobspace blob, the same column in all tables must be in the same blobspace. If the blob column is a tblspace blob, the same column must be a tblspace blob in all tables.

What Happens to Indexes and Triggers?

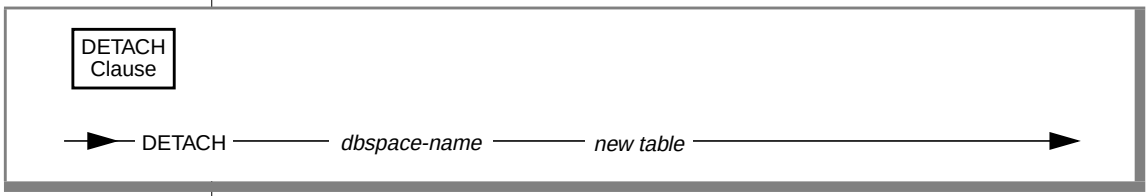
Unless you create separate index fragments, the index fragmentation is the same as the table fragmentation.

When you attach tables, any indexes or triggers that are defined on the consumed table no longer exist, and all rows in the consumed table (**new_acct**) are subject to the indexes and triggers that are defined in the surviving table (**cur_acct**). No triggers are activated with the ATTACH clause, but subsequent data manipulation operations on the "new" rows can fire triggers.

At the end of the ATTACH operation, indexes on the surviving table that were explicitly given a fragmentation strategy remain intact with that fragmentation strategy.

The DETACH Clause

Use the DETACH clause to detach a table fragment from a distribution scheme and place the contents into a new nonfragmented table. See "The FRAGMENT BY Clause for Tables" on page 1-35 for an explanation of distribution schemes.



Element	Purpose	Restrictions	Syntax
dbspace-name	The name of the dbspace that contains the fragment to be detached	The dbspace must exist when you execute the statement.	Identifier, p. 1-723
new table	The table that results after you execute the ALTER FRAGMENT statement	The table must not exist before you execute the statement.	Table Name, p. 1-768

The DETACH clause cannot be applied to a table if that table is the parent of a referential constraint or if a rowid column is defined on the table.

The new table that results from the execution of the DETACH clause does not inherit any indexes or constraints from the original table. Only the data remains.

The following example shows the table **cur_acct** fragmented into two dbspaces, **dbsp1** and **dbsp2**:

```
ALTER FRAGMENT ON TABLE cur_acct DETACH dbsp2 accounts
```

This example detaches **dbsp2** from the distribution scheme for **cur_acct** and places the rows in a new table, **accounts**. Table **accounts** now has the same structure (column names, number of columns, data types, and so on) as table **cur_acct**, but the table **accounts** does not contain any indexes or constraints from the table **cur_acct**. Both tables are now nonfragmented.

The following example shows a table that contains three fragments:

```
ALTER FRAGMENT ON TABLE bus_acct DETACH dbsp3 cli_acct
```

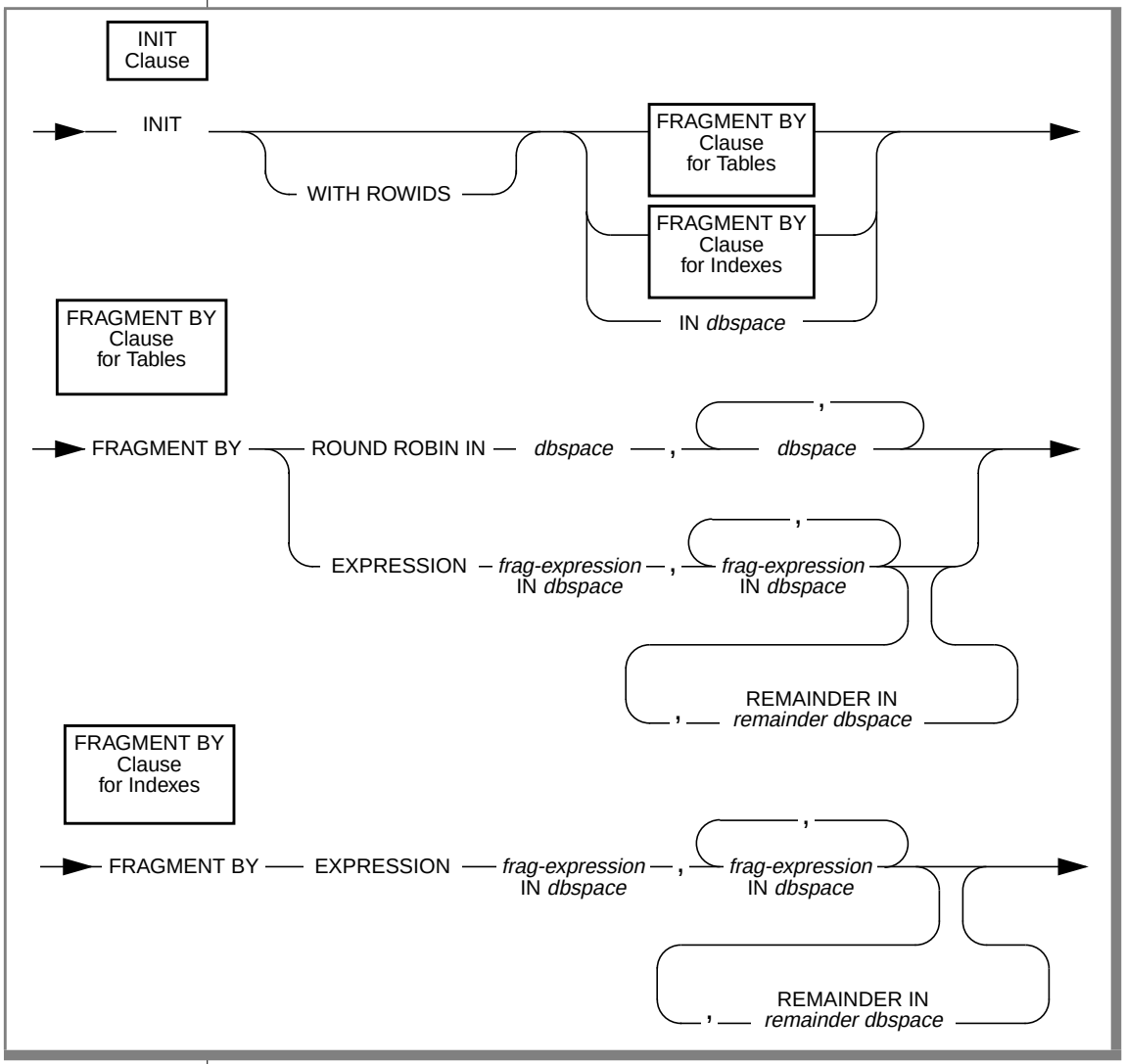
This statement detaches **dbsp3** from the distribution scheme for **bus_acct** and places the rows in a new table, **cli_acct**. Table **cli_acct** now has the same structure (column names, number of columns, data types, and so on) as **bus_acct,** but the table **cli_acct** does not contain any indexes or constraints from the table **bus_acct**. Table **cli_acct** is a nonfragmented table, and table **bus_acct** remains a fragmented table.

The INIT Clause

Use the INIT clause to perform the following functions:

- Change the fragmentation strategy on a single, fragmented table including changing the order of evaluating fragment expressions

- Define and initialize a new fragmentation strategy on a nonfragmented table
- Convert a fragmented table to a nonfragmented table
- Detach an index from a table fragmentation strategy



Element	Purpose	Restrictions	Syntax
dbspace	The dbspace that contains the fragmented information	The dbspace must exist at the time you execute the statement. When you use the FRAGMENT BY clause, you must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces.	Identifier, p. 1-723
frag-expression	An expression that defines a fragment using a range, hash, or arbitrary rule	If you specify a value for *remainder dbspace*, you must specify at least one fragment expression. If you do not specify a value for *remainder dbspace*, you must specify at least two fragment expressions. You can specify a maximum of 2,048 fragment expressions. Each fragment expression can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date / time functions, or aggregates are allowed in *frag-expression*.	Condition, p. 1-643, and Expression, p. 1-671
remainder dbspace	The dbspace that contains data that does not meet the conditions defined in any fragment expression.	If you specify two or more fragment expressions, *remainder dbspace* is optional. If you specify only one fragment expression, *remainder dbspace* is required. The dbspace that is specified in *remainder dbspace* must exist at the time you execute the statement.	Identifier, p. 1-723

The INIT clause allows you to fragment an existing table or index that is not fragmented without redefining the table or index. With the INIT clause, you can also convert an existing fragmentation strategy on a table or index to another fragmentation strategy. Any existing fragmentation strategy is discarded, and records are moved to fragments as defined in the new fragmentation strategy. The INIT clause also allows you to convert a fragmented table or index to a nonfragmented table or index.

When you use the INIT clause to fragment an existing nonfragmented table, all indexes on the table become fragmented in the same way as the table.

Changing an Existing Fragmentation Strategy on a Single Table

You can redefine a fragmentation strategy if you decide that your initial strategy does not fulfill your needs. The following example shows the statement that originally defined the fragmentation strategy on the table **account** and then shows the ALTER FRAGMENT statement that redefines the fragmentation strategy:

```
CREATE TABLE account (col1 int, col2 int)
    FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2;

ALTER FRAGMENT ON TABLE account
    INIT FRAGMENT BY EXPRESSION
    MOD(col1, 3) = 0 in dbsp1,
    MOD(col1, 3) = 1 in dbsp2,
    MOD(col1, 3) = 2 in dbsp3;
```

When you want to redefine a fragmentation strategy, and any existing dbspaces are full, you must fragment the table in different dbspaces than the full dbspaces.

Fragmenting Unique and System Indexes

You can fragment unique indexes only if the table uses an expression-based distribution scheme. The columns that are referenced in the fragment expression must be indexed columns. If your ALTER FRAGMENT INIT statement fails to meet either of these restrictions, the INIT fails, and work is rolled back.

You might have an attached unique index on a table fragmented by Column A. If you use INIT to change the table fragmentation to Column B, the INIT fails because the unique index is defined on Column A. To resolve this issue, you can use the INIT clause on the index to detach it from the table fragmentation strategy and fragment it separately.

System indexes (such as those used in referential constraints and unique constraints) utilize user indexes if the indexes exist. If no user indexes can be utilized, system indexes remain nonfragmented and are moved to the dbspace where the database was created. To fragment a system index, create the fragmented index on the constraint columns, and then use the ALTER TABLE statement to add the constraint.

Converting a Fragmented Table to a Nonfragmented Table

You might decide that you no longer want a table to be fragmented. You can use the INIT clause to convert a fragmented table to a nonfragmented table. The following example shows the original fragmentation definition as well as how to use the ALTER FRAGMENT statement to convert the table:

```
CREATE TABLE checks (col1 int, col2 int)
    FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2, dbsp3;

ALTER FRAGMENT ON TABLE checks INIT IN dbsp1;
```

You must use the IN *dbspace* clause to place the table in a dbspace explicitly.

When you use the INIT clause to change a fragmented table to a nonfragmented table (that is, to rid the table of any fragmentation strategy), all indexes that are fragmented in the same way as the table become nonfragmented indexes. Similarly, system indexes that do not utilize user indexes become nonfragmented indexes and are moved from the database dbspace to the table dbspace. If any system indexes use detached user indexes, the system indexes are not affected by the use of the INIT clause on the table.

Defining a Fragmentation Strategy on a Nonfragmented Table

You can use the INIT clause to define a fragmentation strategy on a nonfragmented table. It does not matter whether the table was created with a storage option. The following example shows the original table definition as well as how to use the ALTER FRAGMENT statement to fragment the table:

```
CREATE TABLE balances (col1 int, col2 int) IN dbsp1;

ALTER FRAGMENT ON TABLE balances INIT
    FRAGMENT BY EXPRESSION
    col1 <= 500 IN dbsp1,
    col1 > 500 and col1 <=1000 IN dbsp2,
    REMAINDER IN dbsp3;
```

Detaching an Index from a Table-Fragmentation Strategy

You can detach an index from a table-fragmentation strategy with the INIT clause, which causes an attached index to become a detached index. This breaks any dependency of the index on the table fragmentation strategy.

The WITH ROWIDS Clause

Nonfragmented tables contain a pseudocolumn called the rowid column. Fragmented tables do not contain this column unless it is explicitly created.

Use the WITH ROWIDS clause to add a new column called the rowid column. OnLine assigns a unique number to each row that remains stable for the existence of the row. The database server creates an index that it uses to find the physical location of the row. Each row contains an additional 4 bytes to store the rowid column after you add the WITH ROWIDS clause.

Important: *Informix recommends that you use primary keys, rather than the rowid column, as an access method.*

The FRAGMENT BY Clause for Tables

Use the FRAGMENT BY clause for tables to define the distribution scheme, which is either round-robin or expression based.

In a round-robin distribution scheme, specify at least two dbspaces where the fragments are placed. As records are inserted into the table, they are placed in the first available dbspace. OnLine balances the load between the specified dbspaces as you insert records and distributes the rows so that the fragments always maintain approximately the same number of rows. In this distribution scheme, the database server must scan all fragments when it searches for a row.

In an expression-based distribution scheme, each fragment expression in a *rule* specifies a dbspace. The rule specifies how you determine the fragment into which rows are placed. Each fragment expression within the rule isolates data and aids the database server in searching for rows. You can specify one of the following rules:

- Range rule

  A range rule specifies fragment expressions that use a range to specify which rows are placed in a fragment, as shown in the following example:

```
...
FRAGMENT BY EXPRESSION
c1 < 100 IN dbsp1,
c1 >= 100 and c1 < 200 IN dbsp2,
c1 >= 200 IN dbsp3;
```

- Hash rule

  A hash rule specifies fragment expressions that are created when you use a hash algorithm, which is often implemented with the MOD function, as shown in the following example:

```
.
.
.
FRAGMENT BY EXPRESSION
MOD(id_num, 3) = 0 IN dbsp1,
MOD(id_num, 3) = 1 IN dbsp2,
MOD(id_num, 3) = 2 IN dbsp3;
```

- Arbitrary rule

  An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically includes the use of OR clauses to group data, as shown in the following example:

```
.
.
.
FRAGMENT BY EXPRESSION
zip_num = 95228 OR zip_num = 95443 IN dbsp2,
zip_num = 91120 OR zip_num = 92310 IN dbsp4,
REMAINDER IN dbsp5;
```

Warning: *When you define the distribution scheme for a table and specify a date value in a fragment expression, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

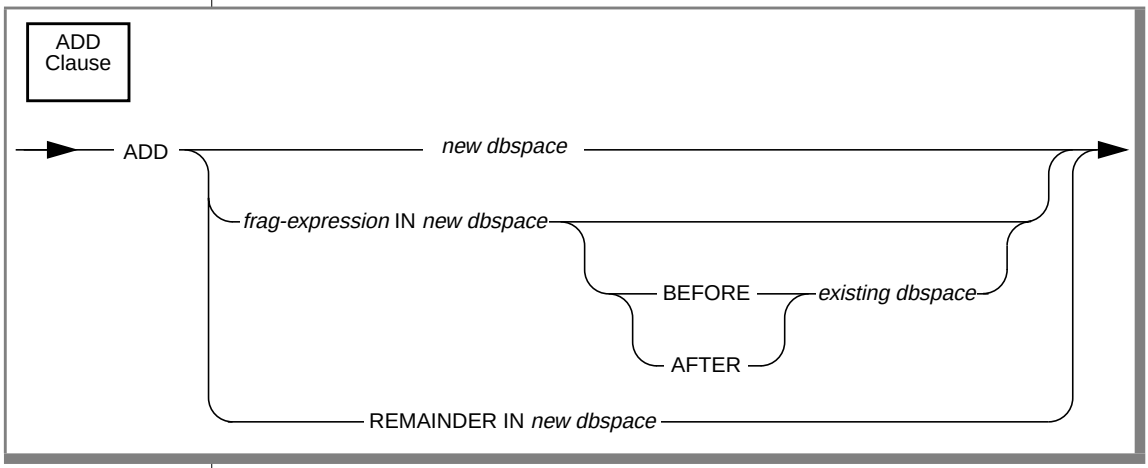
The FRAGMENT BY Clause for Indexes

Use the FRAGMENT BY clause for indexes to define the expression-based distribution scheme. Like the FRAGMENT BY clause for tables, the FRAGMENT BY clause for indexes supports range rules, hash rules, and arbitrary rules. See "The FRAGMENT BY Clause for Tables" on page 1-35 for an explanation of these rules.

Warning: *When you define the distribution scheme for an index and specify a date value in a fragment expression, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

The ADD Clause

Use the ADD clause to add another fragment to an existing fragmentation list.



Element	Purpose	Restrictions	Syntax
existing dbspace	A dbspace name specified in an existing fragmentation list	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-723
frag-expression	The range, hash, or arbitrary expression that defines the added fragment	The *frag-expression* can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date/time functions, or aggregates are allowed in *frag-expression*.	Condition, p. 1-643, and Expression, p. 1-671
new dbspace	The added dbspace in a round-robin distribution scheme	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-723

Adding a New Dbspace to a Round-Robin Distribution Scheme

You can add more dbspaces to a round-robin distribution scheme. The following example shows the original round-robin definition:

```
CREATE TABLE book (col1 int, col2 title)
FRAGMENT BY ROUND ROBIN in dbsp1, dbsp4;
```

To add another dbspace, use the ADD clause, as shown in the following example:

```
ALTER FRAGMENT ON TABLE book ADD dbsp3;
```

Adding Fragment Expressions

Adding a fragment expression to the fragmentation list in an expression-based distribution scheme can shuffle records from some existing fragments into the new fragment. When you add a new fragment into the middle of the fragmentation list, all the data existing in fragments after the new one must be re-evaluated. The following example shows the original expression definition:

```
.
.
.
    FRAGMENT BY EXPRESSION
    c1 < 100 IN dbsp1,
    c1 >= 100 and c1 < 200 IN dbsp2,
    REMAINDER IN dbsp3;
```

If you want to add another fragment to the fragmentation list and have this fragment hold rows between 200 and 300, use the following ALTER FRAGMENT statement:

```
ALTER FRAGMENT ON TABLE news ADD
    c1 >= 200 and c1 < 300 IN dbsp4;
```

Any rows that were formerly in the remainder fragment and that fit the criteria c1 >=200 and c1 < 300 are moved to the new dbspace.

Warning: *When you specify a date value in a fragment expression, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

The BEFORE and AFTER Clauses

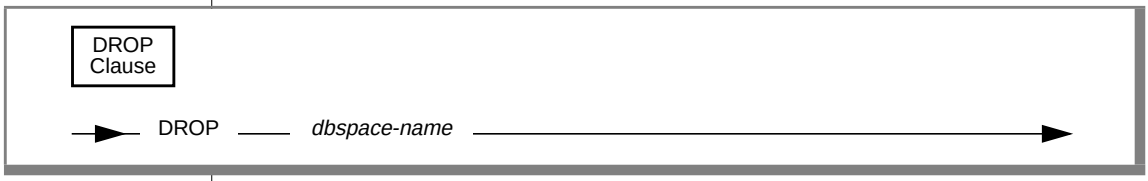
The BEFORE and AFTER clauses allow you to place a new fragment in a dbspace either before or after an existing dbspace. Use the BEFORE and AFTER clauses only when the distribution scheme is expression based (not round-robin). You cannot add a new fragment after the remainder fragment. Adding a new fragment without an explicit BEFORE or AFTER clause places the added fragment at the end of the fragmentation list. However, if the fragmentation list contains a REMAINDER clause, the added fragment is added before the remainder fragment (that is, the remainder remains the last item on the fragment list).

The REMAINDER Clause

You cannot add a remainder fragment when one already exists. When you add a new fragment to the end of the fragmentation list, and a remainder fragment exists, the records in the remainder fragment are retrieved and re-evaluated. These records can be moved to the new fragment. The remainder fragment always remains the last item in the fragment list.

The DROP Clause

Use the DROP clause to drop an existing fragment from a fragmentation list.



Element	Purpose	Restrictions	Syntax
dbspace-name	The name of the dbspace that contains the dropped fragment	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-723

You cannot drop one of the fragments when the table contains only two fragments. You cannot drop a fragment in a table that is fragmented with an expression-based distribution scheme if the fragment contains data that cannot be moved to another fragment. If the distribution scheme contains a REMAINDER clause, or if the expressions were constructed in an overlapping manner, you can drop a fragment that contains data.

When you want to make a fragmented table nonfragmented, use either the INIT or DETACH clause.

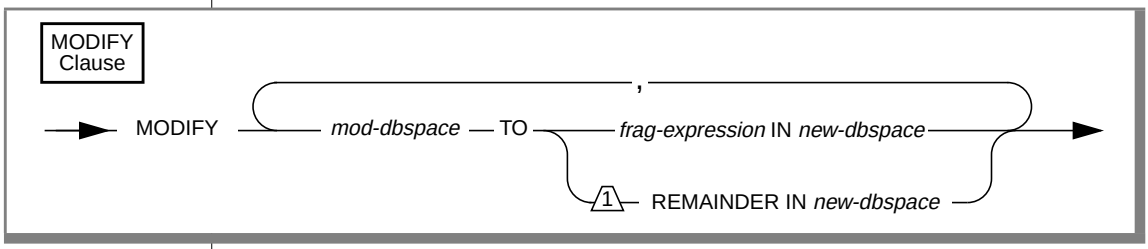
When you drop a fragment from a dbspace, the underlying dbspace is not affected. Only the fragment data within that dbspace is affected. When you drop a fragment all the records located in the fragment move to another fragment. The destination fragment might not have enough space for the additional records. When this happens, follow one of the procedures that are listed in "Making More Space" on page 1-24 to increase your space, and retry the procedure.

The following examples show how to drop a fragment from a fragmentation list. The first line shows how to drop an index fragment, and the second line shows how to drop a table fragment.

```
ALTER FRAGMENT ON INDEX cust_indx DROP dbsp2;

ALTER FRAGMENT ON TABLE customer DROP dbsp1;
```

The MODIFY Clause

Use the MODIFY clause to change an existing fragment expression on an existing dbspace. You can also use the MODIFY clause to move a fragment expression from one dbspace to a different dbspace.



Element	Purpose	Restrictions	Syntax
frag-expression	The modified range, hash, or arbitrary expression	The fragment expression can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date/time functions, or aggregates are allowed in *frag-expression*.	Condition, p. 1-643, and Expression, p. 1-671
mod-dbspace	The modified dbspace	The dbspace must exist when you execute the statement.	Identifier, p. 1-723
new-dbspace	The dbspace that contains the modified information	The dbspace must exist when you execute the statement.	Identifier, p. 1-723

General Usage

When you use the MODIFY clause, the underlying dbspaces are not affected. Only the fragment data within the dbspaces is affected.

You cannot change a REMAINDER fragment into a nonremainder fragment if records within the REMAINDER fragment do not pass the new expression.

Warning: *When you specify a date value in a fragment expression in the MODIFY clause, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

Changing the Expression in an Existing Dbspace

When you use the MODIFY clause to change an expression without changing the dbspace storage for the expression, you must use the same name for the *mod dbspace* and the *new dbspace*.

The following example shows how to use the MODIFY clause to change an existing expression:

```
ALTER FRAGMENT ON TABLE cust_acct
    MODIFY dbsp1 to acct_num < 65 IN dbsp1
```

Moving an Expression from One Dbspace to Another

When you use the MODIFY clause to move an expression from one dbspace to another, *mod-dbspace* is the name of the dbspace where the expression was previously located, and *new-dbspace* is the new location for the expression.

The following example shows how to use the MODIFY clause to move an expression from one dbspace to another:

```
ALTER FRAGMENT ON TABLE cust_acct
    MODIFY dbsp1 to acct_num < 35 in dbsp2
```

In this example, the distribution scheme for the **cust_acct** table is modified so that all row items in the column **acct_num** that are less than 35 are now contained in the dbspace **dbsp2**. These items were formerly contained in the dbspace **dbsp1**.

Changing the Expression and Moving it to a New Dbspace

When you use the MODIFY clause to change the expression and move it to a new dbspace, change both the expression name and the dbspace name.

What Happens to Indexes?

If your indexes are attached indexes, and you modify the table, the index fragmentation strategy is also modified.

References

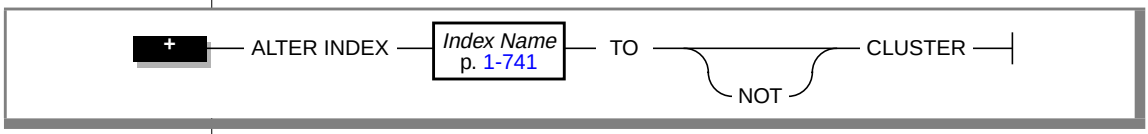
See the CREATE TABLE, CREATE INDEX, ALTER TABLE statements in this manual. Also see the Condition, Data Type, Expression, and Identifier segments.

For a task-oriented discussion of each clause in the ALTER FRAGMENT statement, see Chapter 9 of the *Informix Guide to SQL: Tutorial*.

ALTER INDEX

Use the ALTER INDEX statement to put the data in a table in the order of an existing index or to release an index from the clustering attribute.

Syntax



Usage

The ALTER INDEX statement works only on indexes that are created with the CREATE INDEX statement; it does not affect constraints that are created with the CREATE TABLE statement.

SE

You cannot use a ROLLBACK WORK statement to undo an ALTER INDEX statement. When you roll back a transaction that contains an ALTER INDEX statement, the index remains altered; you do not receive an error message.

When you have an audit trail on the table, you cannot use the ALTER INDEX statement. When you want to change an index on an audited table, you must first drop the audit on the table, alter the index, and create a new audit for the table. ♦

You cannot alter the index of a temporary table.

The TO CLUSTER Option

The TO CLUSTER option causes the rows in the physical table to reorder in the indexed order.

The following example shows how you can use the ALTER INDEX TO CLUSTER statement to order the rows in the **orders** table physically. The CREATE INDEX statement creates an index on the **customer_num** column of the table. Then the ALTER INDEX statement causes the physical ordering of the rows.

```
CREATE INDEX ix_cust ON orders (customer_num);

ALTER INDEX ix_cust TO CLUSTER;
```

Reordering causes rewriting the entire file. This process can take a long time, and it requires sufficient disk space to maintain two copies of the table.

While a table is clustering, the table is locked IN EXCLUSIVE MODE. When another process is using the table to which *index name* belongs, the database server cannot execute the ALTER INDEX statement with the TO CLUSTER option; it returns an error unless lock mode is set to WAIT. (When lock mode is set to WAIT, the database server retries the ALTER INDEX statement.)

Over time, if you modify the table, you can expect the benefit of an earlier cluster to disappear because rows are added in space-available order, not sequentially. You can recluster the table to regain performance by issuing another ALTER INDEX TO CLUSTER statement on the clustered index. You do not need to drop a clustered index before you issue another ALTER INDEX TO CLUSTER statement on a currently clustered index.

The TO NOT CLUSTER Option

The NOT option drops the cluster attribute on the *index name* without affecting the physical table. Because only one clustered index per table can exist, you must use the NOT option to release the cluster attribute from one index before you assign it to another. The following statements illustrate how to remove clustering from one index and how a second index physically reclusters the table:

```
CREATE UNIQUE INDEX ix_ord
    ON orders (order_num);

CREATE CLUSTER INDEX ix_cust
    ON orders (customer_num);
.
.
.

ALTER INDEX ix_cust TO NOT CLUSTER;

ALTER INDEX ix_ord TO CLUSTER;
```

The first two statements create indexes for the **orders** table and cluster the physical table in ascending order on the **customer_num** column. The last two statements recluster the physical table in ascending order on the **order_num** column.

References

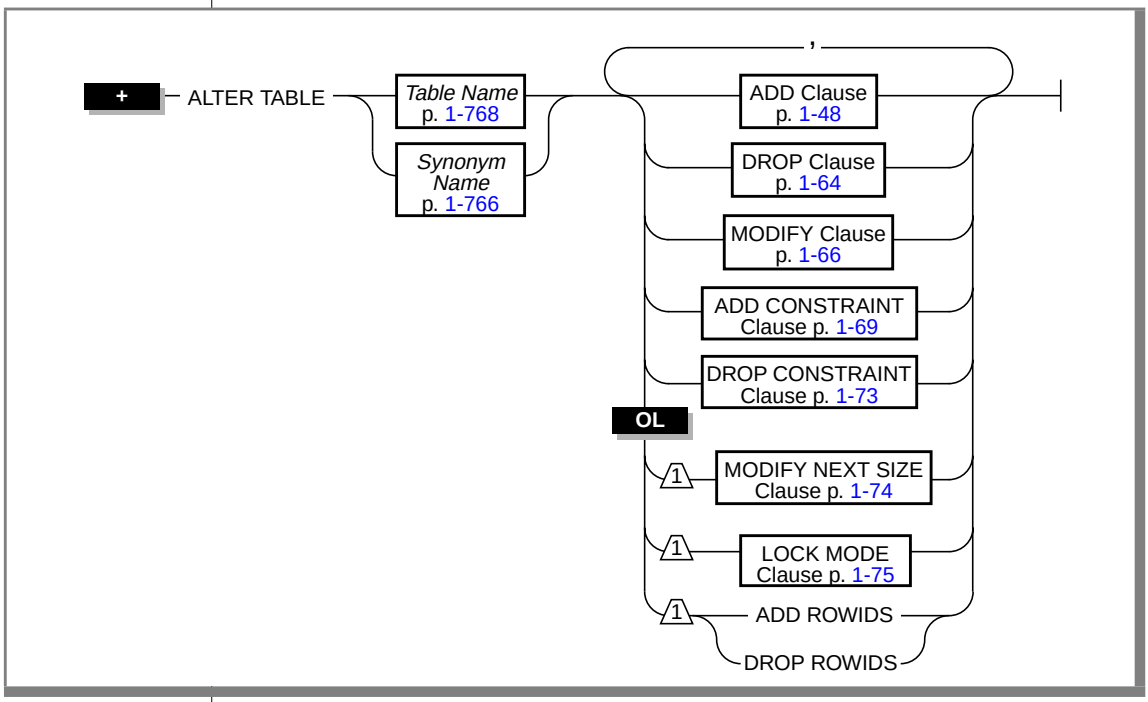
See the CREATE INDEX and CREATE TABLE statements in this chapter.

In the *INFORMIX-OnLine Dynamic Server Performance Guide*, see the discussion of clustered indexes.

ALTER TABLE

Use the ALTER TABLE statement to add a column to or delete a column from a table, modify the data constraints that are placed on a column, add a constraint to a column or a composite list of columns, drop a constraint that is associated with a column or a composite list of columns, or change the extent size. You can also use the ALTER TABLE statement to add a rowid column to a fragmented table or drop a rowid column from a fragmented table.

Syntax



Usage

To use the ALTER TABLE statement, you must meet one of the following conditions:

- You must have the DBA privilege on the database where the table resides.
- You must own the table.
- You must have the Alter privilege on the specified table and the Resource privilege on the database where the table resides.

You cannot alter a temporary table.

To add a referential constraint, you must have the DBA or References privilege on either the referenced columns or the referenced table.

When you add a constraint of any type, the name of the constraint must be unique within the database.

ANSI

When you add a constraint of any type, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

To drop a constraint in a database, you must have the DBA privilege or be the owner of the constraint. If you are the owner of the constraint but not the owner of the table, you must have Alter privilege on the specified table. You do not need the References privilege to drop a constraint.

Altering a table on which a view depends might invalidate the view.

You can use one or more of the ADD, DROP, MODIFY, ADD CONSTRAINT, or DROP CONSTRAINT clauses, and you can place them in any order. You can use only one MODIFY NEXT SIZE, LOCK MODE, ADD ROWIDS, or DROP ROWIDS clause. The actions are performed in the order that is specified. If any of the actions fails, the entire operation is cancelled.

SE

You cannot use a ROLLBACK WORK statement to undo an ALTER TABLE statement. When you roll back a transaction that contains an ALTER TABLE statement, the table remains altered; you do not receive an error message. ♦

The ADD ROWIDS and DROP ROWIDS clauses apply specifically to fragmented tables.

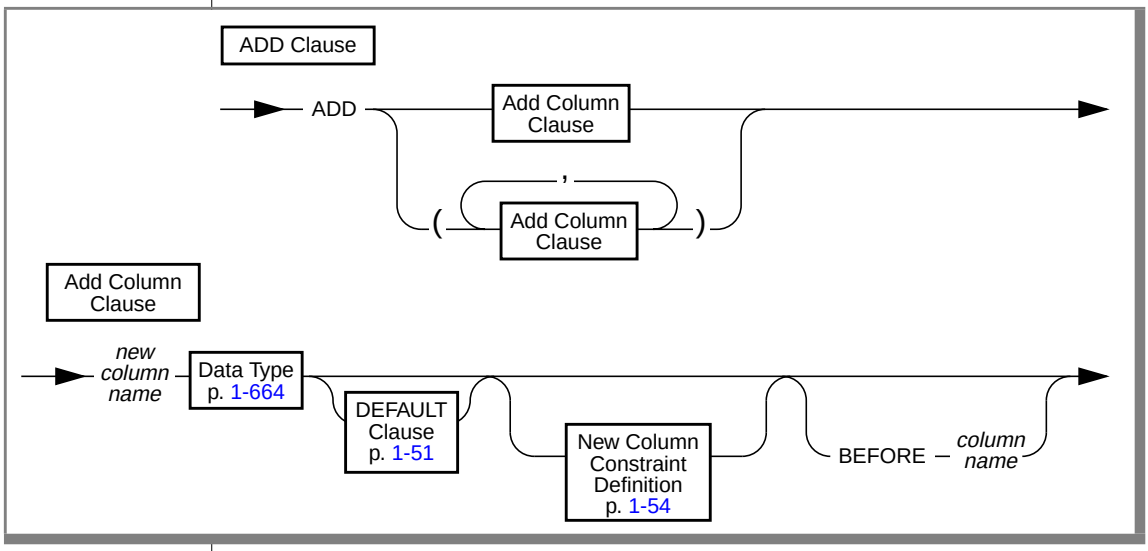
Restrictions for Violations and Diagnostics Tables

Keep the following considerations in mind when you use the ALTER TABLE statement in connection with violations and diagnostics tables:

- You cannot add, drop, or modify a column if the table that contains the column has violations and diagnostics tables associated with it.
- You cannot alter a violations or diagnostics table.
- You cannot add a constraint to a violations or diagnostics table.

See the START VIOLATIONS TABLE statement on page 1-584 for further information on violations and diagnostics tables.

ADD Clause



Element	Purpose	Restrictions	Syntax
column name	The name of a column before which the new column is to be placed	The column must already exist in the table.	Identifier, p. 1-723
new column name	The name of the column that you are adding	You cannot add a SERIAL column if the table contains data.	Identifier, p. 1-723

Use the ADD clause to add a column to a table. You cannot add a SERIAL column to a table if the table contains data.

Algorithms for Adding Columns to Tables

INFORMIX-OnLine Dynamic Server uses the following two algorithms for adding columns to tables:

- If you execute an ALTER TABLE statement that adds a column or list of columns to the end of a table, the database server uses an algorithm that is known as the in-place alter algorithm. When it uses this algorithm, the server allows the table definition to be altered without making the table unavailable to users for longer than the time it takes to update the table definition. Furthermore, the physical addition of the new columns to the table definition occurs essentially in place as rows are updated, without requiring a second copy of the table to be created.
- If you execute an ALTER TABLE statement that does not add a column or list of columns to the end of a table, the database server uses a slower algorithm. When it uses this slower algorithm, the database server performs the alter operation by placing an exclusive lock on the table while it copies the table to be altered to a new table that contains the new table definition. After the copy operation is complete, the database server drops the older version of the table.

Tip: *To add a column to the end of a table, omit the BEFORE option from the ADD clause. When you do not specify a column before which the new column is to be added, the database server adds the new column to the end of the table by default.*

Scope of the In-Place Alter Algorithm

The database server uses the in-place alter algorithm if you specify the ADD clause without the BEFORE option and if you specify any clauses other than the following:

- The DROP clause
- A MODIFY clause that changes the data type of a column or changes the number of characters in a character column

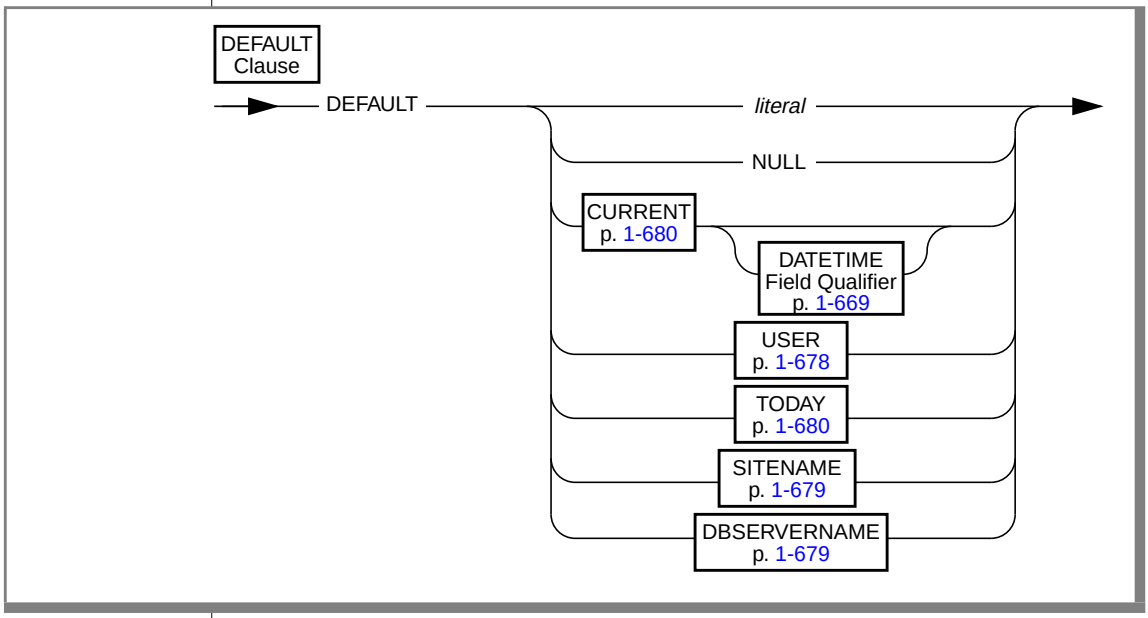
Benefits of the In-Place Alter Algorithm

The in-place alter algorithm lets you alter tables in place instead of creating a new table with the latest table definition and copying rows from the original table to the new table. The in-place alter method reduces the space that is required for altering tables and also increases the availability of the tables that are being altered.

The database server uses the slower algorithm for altering tables whenever your ALTER TABLE statement does not match the conditions for using the in-place alter algorithm. The database server uses the slower algorithm under the following conditions:

- The database server uses the slower algorithm if you specify an ADD clause with the BEFORE option.
- The database server uses the slower algorithm if you specify an ADD clause without the BEFORE option, but you also specify one of the following clauses:
  - The DROP clause
  - A MODIFY clause that changes the data type of a column or changes the number of characters in a character column

DEFAULT Clause



Element	Purpose	Restrictions	Syntax
literal	A literal term that defines alpha or numeric constant characters to be used as the default value for the column	Term must be appropriate type for the column. See "Literal Terms" on page 1-52.	Expression, p. 1-671

You specify a default value to insert into the column when you do not specify an explicit value. When a default is not specified, and the column allows nulls, the default is NULL. When you designate NULL as the default value for a column, you cannot place a not null constraint on the column.

You cannot place a default on SERIAL columns.

When the altered table already has rows in it, the new column contains the default value for all existing rows.

Literal Terms

You can designate *literal* terms as default values. Use a literal term to define alpha or numeric constant characters. To use a literal term as a default value, you must adhere to the rules in the following table.

Use a Literal	With Columns of Data Type
INTEGER	INTEGER, SMALLINT, DECIMAL, MONEY, FLOAT, SMALLFLOAT
DECIMAL	DECIMAL, MONEY, FLOAT, SMALLFLOAT
CHARACTER	CHAR, NCHAR, NVARCHAR, VARCHAR, DATE
INTERVAL	INTERVAL
DATETIME	DATETIME

Characters must be enclosed in quotation marks. Date literals must be formatted in accordance with the DBDATE environment variable. When DBDATE is not set, the format *mm/dd/yyyy* is assumed.

For information on using a literal INTERVAL, see the Literal INTERVAL segment on page 1-749. For more information on using a literal DATETIME, see the Literal DATETIME segment on page 1-746.

Data-Type Requirements

The following table indicates the data type requirements for columns that specify the CURRENT, DBSERVERNAME, SITENAME, TODAY, or USER functions as the default value.

Function Name	Data Type Requirements
CURRENT	DATETIME column with matching qualifier
DBSERVERNAME	CHAR, NCHAR, VARCHAR, or NVARCHAR column at least 18 characters long

(1 of 2)

Function Name	Data Type Requirements
SITENAME	CHAR, NCHAR, VARCHAR, or NVARCHAR column at least 18 characters long
TODAY	DATE column
USER	CHAR column at least 8 characters long

(2 of 2)

Example of a Literal Default Value

The following example adds a column to the **items** table. In **items**, the new column **item_weight** has a literal default value:

```
ALTER TABLE items ADD
    item_weight DECIMAL (6, 2) DEFAULT 2.00 BEFORE total_price
```

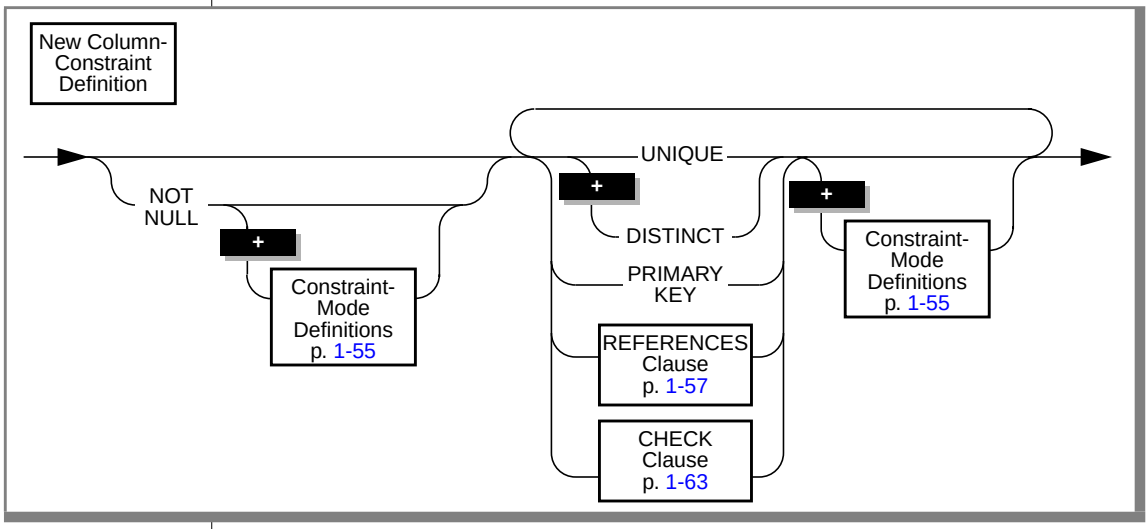
In this example, each existing row in the **items** table has a default value of `2.00` for the **item_weight c**olumn.

Using Not Null Constraints with ADD

When you do not indicate a default value for a column, the default is null unless you place a not null constraint on the column. In this case, if the not null constraint is used, no default value exists for the column, and the column does not allow nulls. When the table contains data, however, you cannot specify a not null constraint when you add a column (unless both the not null constraint and a default value other than null are specified), nor can you specify that the new column has a unique or primary-key constraint. When you want to add a column with a unique constraint, the table can contain a *single* row of data when you issue the ALTER TABLE statement. When you want to add a column with a not null constraint or a primary-key constraint, the table *must* be empty when you issue the ALTER TABLE statement. The following statement is valid only if the **items** table is empty:

```
ALTER TABLE items
    ADD (item_weight DECIMAL(6,2) NOT NULL
        BEFORE total_price)
```

New Column-Constraint Definition



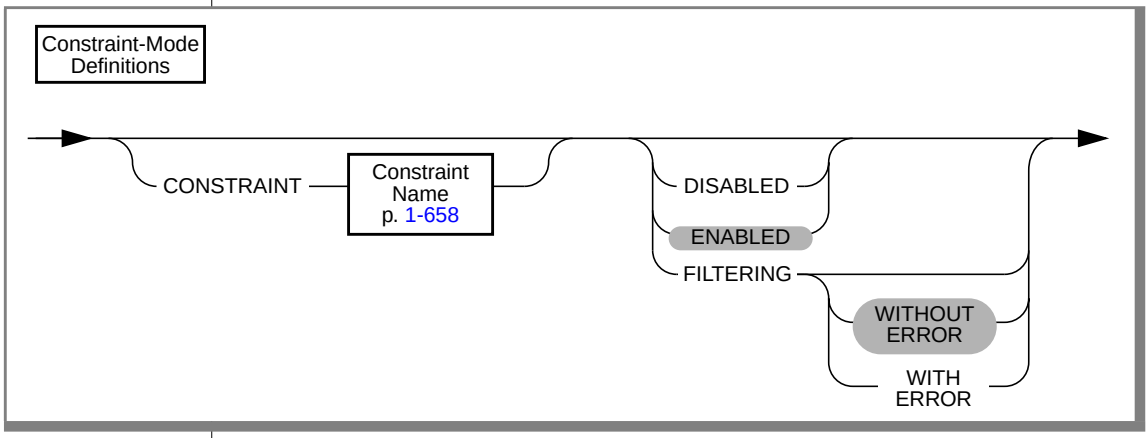
You cannot specify a unique or primary-key constraint on a new column if the table contains data. However, in the case of a unique constraint, the table can contain a *single* row of data. When you want to add a column with a primary-key constraint, the table must be empty when you issue the ALTER TABLE statement.

The following rules apply when you place unique or primary-key constraints on existing columns:

- When you place a unique or primary-key constraint on a column or set of columns, and a unique index already exists on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before you add the constraint.
- When you place a unique or primary-key constraint on a column or set of columns, and a referential constraint already exists on that column or set of columns, the duplicate index is upgraded to unique (if possible), and the index is shared.

You cannot have a unique constraint on a BYTE or TEXT column, nor can you place referential or check constraints on these types of columns. You can place a check constraint on a BYTE or TEXT column. However, you can check only for IS NULL, IS NOT NULL, or LENGTH.

Constraint-Mode Definitions



You can use the Constraint-Mode Definitions option for the following purposes:

- You can assign a name to a constraint on a column.
- You can set a constraint to one of the following object modes: disabled, enabled, or filtering.

Description of Constraint Modes

You can set constraints to the following modes: disabled, enabled, or filtering. These modes are described in the following table.

Constraint Mode	Effect
disabled	A constraint that is created in disabled mode is not enforced during insert, delete, and update operations.
enabled	A constraint that is created in enabled mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement fails.
filtering	A constraint that is created in filtering mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement continues processing, but the bad row is written to the violations table that is associated with the target table. Diagnostic information about the constraint violation is written to the diagnostics table that is associated with the target table.

If you chose the filtering mode, you can specify the WITHOUT ERROR options. The following table describes these options.

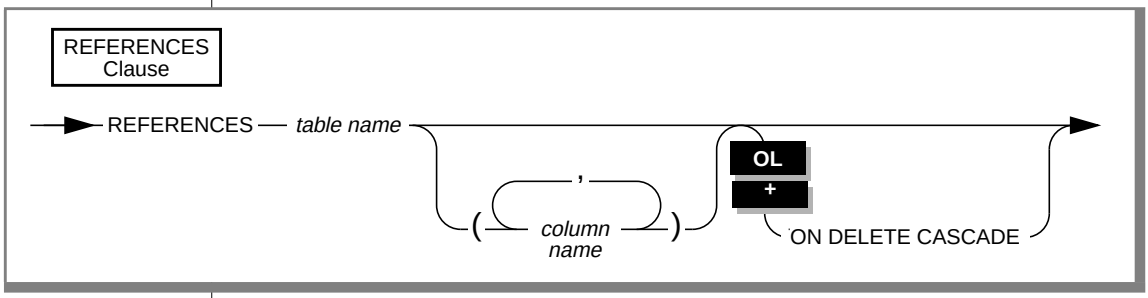
Error Option	Effect
WITHOUT ERROR	When a filtering mode constraint is violated during an insert, delete, or update operation, no integrity-violation error is returned to the user.
WITH ERROR	When a filtering mode constraint is violated during an insert, delete, or update operation, an integrity-violation error is returned to the user.

Using Constraint Modes

You must observe the following rules when you use constraint modes:

- If you do not specify the object mode of a column-level or table-level constraint explicitly, the default mode is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option for a filtering mode constraint, the default error option is WITHOUT ERROR.
- When you add a column-level or table-level constraint to a table and specify the disabled object mode for the constraint, your ALTER TABLE statement succeeds even if existing rows in the table violate the constraint.
- When you add a column-level or table-level constraint to a table and specify the enabled or filtering object mode for the constraint, your ALTER TABLE statement succeeds if no existing rows in the table violate the new constraint. However, if any existing rows in the table violate the constraint, your ALTER TABLE statement fails and returns an error.
- When you add a column-level or table-level constraint to a table in the enabled or filtering object mode, and existing rows in the table violate the constraint, erroneous rows in the base table are not filtered to the violations table. Thus, you cannot use a violations table to detect the erroneous rows in the base table.

The REFERENCES Clause



Element	Purpose	Restrictions	Syntax
column name	A referenced column or set of columns in the referenced table. If the referenced table is different from the referencing table, the default is the primary-key column. If the referenced table is the same as the referencing table, there is no default.	You must observe restrictions on the number of columns you can specify, the data type of the columns, and the existing constraints on the columns. See "Restrictions on the Column Name Variable in the REFERENCES Clause of ALTER TABLE" below.	Identifier, p. 1-723
table name	The name of the referenced table	The referenced table can be the same table as the referencing table, or it can be a different table in the same database.	Table Name, p. 1-768

Restrictions on the Column Name Variable in the REFERENCES Clause of ALTER TABLE

Observe the following restrictions on the referenced column (the column or set of columns that you specify in the *column name* variable).

Number of Columns

The following restrictions apply to the number of columns that you can specify in the *column name* variable:

- The number of referenced columns in the referenced table must match the number of referencing columns in the referencing table.
- If you are using the REFERENCES clause within the ADD or MODIFY clauses, you can specify only one column in the *column name* variable.
- If you are using the REFERENCES clause within the ADD CONSTRAINT clause, you can specify one column or multiple columns in the *column name* variable.

- The maximum number of columns and the total length of columns vary with the database server:
  - If you are using INFORMIX-OnLine Dynamic Server, you can specify a maximum of 16 column names. The total length of all the columns cannot exceed 255 bytes.
  - If you are using INFORMIX-SE, you can specify a maximum of 8 column names. The total length of all the columns cannot exceed 120 bytes.

Data Type of the Column

The data type of each referenced column must be identical to the data type of the corresponding referencing column. The only exception is that a referencing column must be INTEGER if the referenced column is SERIAL.

Existing Constraints on the Column

The referenced column or set of columns must be a unique or primary-key column. That is, the referenced column in the referenced table must already have a unique or primary-key constraint placed upon it.

Using the REFERENCES Clause in ALTER TABLE

Use the REFERENCES clause to reference a column or set of columns in another table or the same table. When you are using the ADD or MODIFY clause, you can reference a single column. When you are using the ADD CONSTRAINT clause, you can reference a single column or a set of columns.

The table that is referenced in the REFERENCES clause must reside in the same database as the altered table.

A referential constraint establishes the relationship between columns in two tables or within the same table. The relationship between the columns is commonly called a *parent-child* relationship. For every entry in the child (referencing) columns, a matching entry must exist in the parent (referenced) columns.

The referenced column (parent or primary-key) must be a column that is a unique or primary-key constraint. When you specify a column in the REFERENCES clause that does not meet this criterion, the database server returns an error.

The referencing column (child or foreign key) that you specify in the Add Column clause can contain null or duplicate values, but every value (that is, all foreign-key columns that contain non-null values) in the referencing columns must match a value in the referenced column.

Relationship Between Referencing and Referenced Columns

A referential constraint has a one-to-one relationship between referencing and referenced columns. If the primary key is a set of columns, the foreign key also must be a set of columns that corresponds to the primary key. The following example creates a new column in the **cust_calls** table, **ref_order**. The **ref_order** column is a foreign key that references the **order_num** column in the **orders** table.

```
ALTER TABLE cust_calls ADD
    ref_order INTEGER
    REFERENCES orders (order_num) BEFORE user_id
```

When you reference a primary key in another table, you do not have to explicitly state the primary-key columns in that table. Referenced tables that do not specify the referenced column default to the primary-key column. In the previous example, because **order_num** is the primary key in the **orders** table, you do not have to reference that column explicitly.

When you place a referential constraint on a column or set of columns, and a duplicate or unique index already exists on that column or set of columns, the index is shared.

The data types of the referencing and referenced column must be identical, unless the primary-key column is SERIAL data type. When you add a column that references a SERIAL column, the column that you are adding must be an INTEGER column.

Using the ON DELETE CASCADE Clause

Cascading deletes allow you to specify whether you want rows deleted in the child table when rows are deleted in the parent table. Normally, you cannot delete data in the parent table if child tables are associated with it. You can decide whether you want the rows in the child table deleted with the ON DELETE CASCADE clause. With the ON DELETE CASCADE clause (or cascading deletes), when you delete a row in the parent table, any rows that are associated with that row (foreign keys) in a child table are also deleted. The principal advantage to the cascading-deletes feature is that it allows you to reduce the quantity of SQL statements you need to perform delete actions.

For example, the **stock** table contains the **stock_num** column as a primary key. The **catalog** table refers to the **stock_num** column as a foreign key. The following ALTER TABLE statements drop an existing foreign-key constraint (without cascading delete) and add a new constraint that specifies cascading deletes:

```
ALTER TABLE catalog DROP CONSTRAINT aa

ALTER TABLE catalog ADD CONSTRAINT
    (FOREIGN KEY (stock_num, manu_code) REFERENCES stock
    ON DELETE CASCADE CONSTRAINT ab)
```

With cascading deletes specified on the child table, in addition to deleting a stock item from the **stock** table, the delete cascades to the **catalog** table that is associated with the **stock_num** foreign key. Of course, this cascading delete works only if the **stock_num** that you are deleting has not been ordered; otherwise, the constraint from the **items** table would disallow the cascading delete. For more information, see "What Happens to Multiple Child Tables?" on page 1-62.

You specify cascading deletes with the REFERENCES clause on the ADD CONSTRAINT clause. You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to specify cascading deletes in tables; however, you do need the Delete privilege on tables that are referenced in the DELETE statement. After you indicate cascading deletes, when you delete a row from a parent table, OnLine deletes any associated matching rows from the child table.

Use the ADD CONSTRAINT clause to add a REFERENCES clause with the ON DELETE CASCADE clause constraint.

What Happens to Multiple Child Tables?

When you have a parent table with two child tables, one with cascading deletes specified and the other without cascading deletes, and you attempt to delete a row from the parent table that applies to both child tables, the delete statement fails, and no rows are deleted from either the parent or child tables.

In the previous example, the **stock** table is also parent to the **items** table. However, you do not need to add the cascading-delete clause to the **items** table if you are planning to delete only unordered items. The **items** table is used only for ordered items.

Locking and Logging

During deletes, the database server places locks on all qualifying rows of the referenced and referencing tables. You must turn logging on when you perform the deletes. When logging is turned off in a database, even temporarily, deletes do not cascade. This restriction applies because you have no way to roll back actions if logging is turned off . For example, if a parent row is deleted, and the system crashes before the child rows are deleted, the database would have dangling child records. Such records would violate referential integrity. However, when logging is turned back on, subsequent deletes cascade.

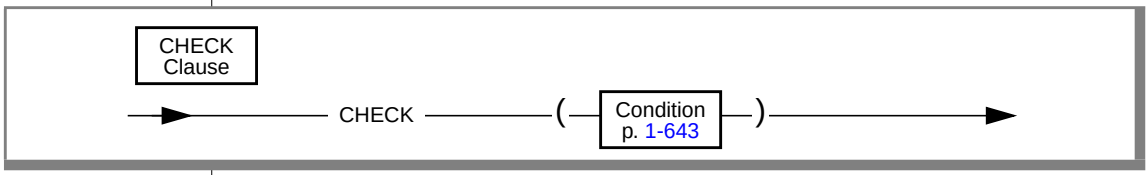
Restriction on Cascading Deletes

Cascading deletes can be used for most deletes except correlated subqueries. In correlated subqueries, the subquery (or inner SELECT) is correlated when the value it produces depends on a value produced by the outer SELECT statement that contains it. If you have implemented cascading deletes, you cannot write deletes that use a child table in the correlated subquery. You receive an error when you attempt to delete from a query that contains such a correlated subquery.

Locks Held During Creation of a Referential Constraint

When you create a referential constraint, an exclusive lock is placed on the referenced table. The lock is released after you finish with the ALTER TABLE statement or at the end of a transaction (if you are altering a table in a database with transactions, and you are using transactions).

CHECK Clause



A check constraint designates a condition that must be met *before* data can be inserted into a column. If a row evaluates to false for any check constraint that is defined on a table during an insert or update, the database server returns an error.

Check constraints are defined using *search conditions*. The search condition cannot contain the following items: subqueries, aggregates, host variables, rowids, or stored procedure calls. In addition, the search condition cannot contain the following functions: the CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY functions.

Warning: *When you specify a date value in a search condition, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on how the database server interprets the search condition. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect how the database server interprets the search condition, so the check constraint may not work as you intended. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

You cannot create check constraints for columns across tables. When you are using the ADD or MODIFY clause, the check constraint cannot depend upon values in other columns of the same table. The following example adds a new column, **unit_price**, to the **items** table and includes a check constraint that ensures that the entered value is greater than 0:

```
ALTER TABLE items
    ADD (unit_price MONEY (6,2) CHECK (unit_price > 0) )
```

To create a constraint that checks values in more than one column, use the ADD CONSTRAINT clause. The following example builds a constraint on the column that was added in the previous example. The check constraint now spans two columns in the table.

```
ALTER TABLE items ADD CONSTRAINT
    CHECK (unit_price < total_price)
```

BEFORE Option

Use the BEFORE option of the ADD clause to specify the column before which a new column or list of columns is to be added. The column that you specify in the BEFORE option must be an existing column in the table.

If you do not include the BEFORE option in the ADD clause, the database server adds the new column or list of columns to the end of the table definition by default.

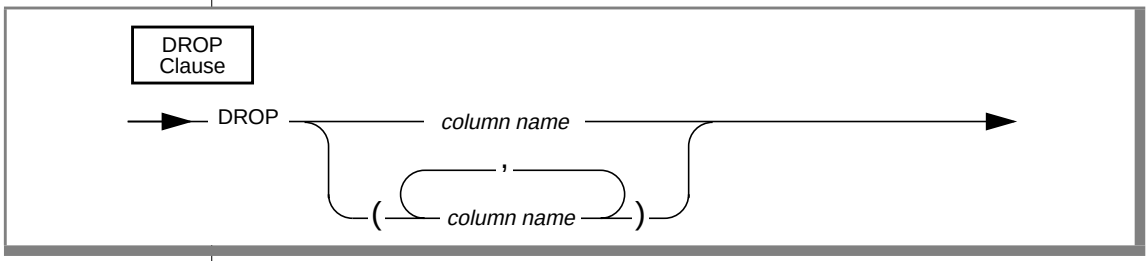
In the following example, to add the **item_weight** column before the **total_price** column, include the BEFORE option in the ADD clause:

```
ALTER TABLE items
    ADD (item_weight DECIMAL(6,2) NOT NULL
        BEFORE total_price)
```

In the following example, to add the **item_weight** column to the end of the table, omit the BEFORE option from the ADD clause:

```
ALTER TABLE items
    ADD (item_weight DECIMAL(6,2) NOT NULL)
```

DROP Clause



Element	Purpose	Restrictions	Syntax
column name	The name of the column that you want to drop	The column must already exist in the table. If the column is referenced in a fragment expression, it cannot be dropped.	Identifier, p. 1-723

Use the DROP clause to drop one or more columns from a table.

How Dropping a Column Affects Constraints

When you drop a column, all constraints placed on that column are dropped, as described in the following list:

- All single-column constraints are dropped.
- All referential constraints that reference the column are dropped.
- All check constraints that reference the column are dropped.
- If the column is part of a multiple-column unique or primary-key constraint, the constraints placed on the multiple columns are also dropped. This action, in turn, triggers the dropping of all referential constraints that reference the multiple columns.

Because any constraints that are associated with a column are dropped when the column is dropped, the structure of other tables might also be altered when you use this clause. For example, if the dropped column is a unique or primary key that is referenced in other tables, those referential constraints also are dropped. Therefore the structure of those other tables is also altered.

How Dropping a Column Affects Triggers

When you drop a column that occurs in the triggering column list of an UPDATE trigger, the column is dropped from the triggering column list. If the column is the only member of the triggering column list, the trigger is dropped from the table. See the CREATE TRIGGER statement on page 1-192 for more information on triggering columns in an UPDATE trigger.

How Dropping a Column Affects Views

When you alter a table by dropping a column, views that depend on the column are not modified. However, if you attempt to use the view, you receive an error message indicating that the column was not found.

Views are not dropped because you can change the order of columns in a table by dropping a column and then adding a new column with the same name. Views based on that table continue to work. They retain their original sequence of columns.

MODIFY Clause

Element **Purpose**

Use the MODIFY clause to change the data type of a column and the length of a character column, to add or change the default value for a column, and to allow or disallow nulls in a column.

When you modify a column, *all* attributes previously associated with that column (that is, default value, single-column check constraint, or referential constraint) are dropped. When you want certain attributes of the column to remain, such as PRIMARY KEY, you must respecify those attributes. For example, if you are changing the data type of an existing column, **quantity**, to SMALLINT, and you want to keep the default value (in this case, 1) and non-null attributes for that column, you can issue the following ALTER TABLE statement:

```
ALTER TABLE items
    MODIFY (quantity SMALLINT DEFAULT '1' NOT NULL)
```

Tip: *Both attributes are specified again in the MODIFY clause.*

When you modify a column that has column constraints associated with it, the following constraints are dropped:

- All single-column constraints are dropped.
- All referential constraints that reference the column are dropped.
- If the modified column is part of a multiple-column unique or primary-key constraint, all referential constraints that reference the multiple columns also are dropped.

For example, if you modify a column that has a unique constraint, the unique constraint is dropped. If this column was referenced by columns in other tables, those referential constraints are also dropped. In addition, if the column is part of a multiple-column unique or primary-key constraint, the multiple-column constraints are not dropped, but any referential constraints placed on the column by other tables *are* dropped. For example, a column is part of a multiple-column primary-key constraint. This primary key is referenced by foreign keys in two other tables. When this column is modified, the multiple-column primary-key constraint is not dropped, but the referential constraints placed on it by the two other tables *are* dropped.

If you modify a column that appears in the triggering column list of an UPDATE trigger, the trigger is unchanged.

Altering the Next Serial Number

You can use the MODIFY clause to reset the next value of a SERIAL column. You cannot set the next value below the current maximum value in the column because that action can cause the database server to generate duplicate numbers. However, you can set the next value to any value higher than the current maximum, which creates gaps in the sequence.

Altering the Structure of Tables

When you use the MODIFY clause, you can also alter the structure of other tables. If the modified column is referenced by other tables, those referential constraints are dropped. You must add those constraints to the referencing tables again, using the ALTER TABLE statement.

When you change the data type of an existing column, all data is converted to the new data type, including numbers to characters and characters to numbers (if the characters represent numbers). The following statement changes the data type of the **quantity** column:

```
ALTER TABLE items MODIFY (quantity CHAR(6))
```

When a unique or primary-key constraint exists, however, conversion takes place only if it does not violate the constraint. If a data-type conversion would result in duplicate values (by changing FLOAT to SMALLFLOAT, for example, or by truncating CHAR values), the ALTER TABLE statement fails.

Modifying Tables for Null Values

You can modify an existing column that formerly permitted nulls to disallow nulls, provided that the column contains no null values. To do this, specify MODIFY with the same *column name* and data type and the NOT NULL keywords. The NOT NULL keywords create a not null constraint on the column.

You can modify an existing column that did not permit nulls to permit nulls. To do this, specify MODIFY with the *column name* and the existing data type, and omit the NOT NULL keywords. The omission of the NOT NULL keywords drops the not null constraint on the column. However, if a unique index exists on the column, you can remove it using the DROP INDEX statement.

An alternative method of permitting nulls in an existing column that did not permit nulls is to use the DROP CONSTRAINT clause to drop the not null constraint on the column.

Adding a Constraint When Existing Rows Violate the Constraint

If you use the MODIFY clause to add a constraint in the enabled mode and receive an error message because existing rows would violate the constraint, you can take the following steps to add the constraint successfully:

1. Add the constraint in the disabled mode.

   Issue the ALTER TABLE statement again, but this time specify the DISABLED keyword in the MODIFY clause.

2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.

3. Issue a SET statement to switch the object mode of the constraint to the enabled mode.

   When you issue this statement, existing rows in the target table that violate the constraint are duplicated in the violations table; however, you receive an integrity-violation error message, and the constraint remains disabled.

4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table.

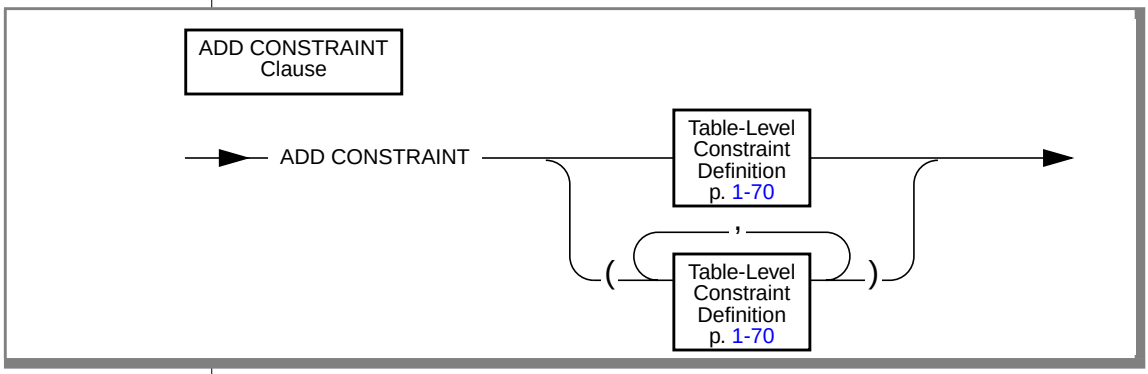
   You might need to join the violations and diagnostics tables to get all the necessary information.

5. Take corrective action on the rows in the target table that violate the constraint.

6. After you fix all the nonconforming rows in the target table, issue the SET statement again to switch the disabled constraint to the enabled mode.

   This time the constraint is enabled, and no integrity-violation error message is returned because all rows in the target table now satisfy the new constraint.

ADD CONSTRAINT Clause



Use the ALTER TABLE statement with the ADD CONSTRAINT keywords to specify a constraint on a new or existing column or on a set of columns. For example, to add a unique constraint to the **fname** and **lname** columns of the **customer** table, use the following statement:

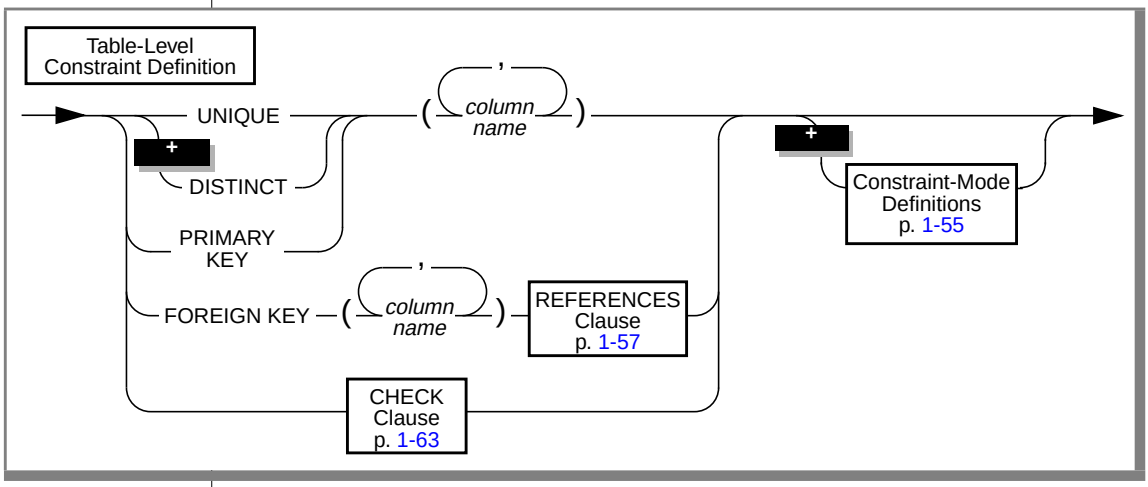
```
ALTER TABLE customer
    ADD CONSTRAINT UNIQUE (lname, fname)
```

To name the constraint, change the preceding statement, as shown in the following example:

```
ALTER TABLE customer
    ADD CONSTRAINT UNIQUE (lname, fname) CONSTRAINT u_cust
```

When you do not provide a constraint name, the database server provides one. You can find the name of the constraint in the **sysconstraints** system catalog table. See Chapter 2 of the *Informix Guide to SQL: Reference* for more information about the **sysconstraints** system catalog table.

Table-Level Constraint Definition



Element	Purpose	Restrictions	Syntax
column name	The name of the column or columns on which the constraint is placed	If you are using OnLine, the maximum number of columns is 16, and the total length of all the columns cannot exceed 255 bytes. If you are using SE, the maximum number of columns is 8, and the total length of all the columns cannot exceed 120 bytes.	Identifier, p. 1-723

Use the Table-Level Constraint Definition option to add a table-level constraint. You can define a table-level constraint on one column or a set of columns. You can assign a name to the constraint and set its object mode by means of the Constraint Mode Definitions option. See "Constraint-Mode Definitions" on page 1-55 for further information.

Adding a Unique Constraint

You must follow certain rules when you add a unique constraint.

The column or columns can contain only unique values.

When you place a unique constraint on a column or set of columns, and a unique index already exists on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the unique constraint.

When you add a unique constraint, the name of the constraint must be unique within the database.

ANSI

When you add a unique constraint, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

A composite list can include no more than 16 column names. The total length of all the columns cannot exceed 255 bytes.

SE

A composite list can include no more than 8 column names, and the total length of all the columns cannot exceed 120 bytes. ♦

Adding a Primary-Key or Unique Constraint

You must follow certain rules when you add a unique or primary-key constraint.

When you place a unique or primary-key constraint on a column or set of columns, and a unique index already exists on that column or set of columns, the constraint shares the index. However, if the existing index allows duplicates, the database server returns an error. You must then drop the existing index before adding the constraint.

When you place a unique or primary-key constraint on a column or set of columns, and a referential constraint already exists on that column or set of columns, the duplicate index is upgraded to unique (if possible) and the index is shared.

When you place a referential constraint on a column or set of columns, and a referential constraint already exists on that column or set of columns, the duplicate index is upgraded to unique (if possible), and the index is shared.

When you add a unique or primary-key constraint, the name of the constraint must be unique within the database.

ANSI

When you add a unique or primary-key constraint, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

A composite list can include no more than 16 column names. The total length of all the columns cannot exceed 255 bytes.

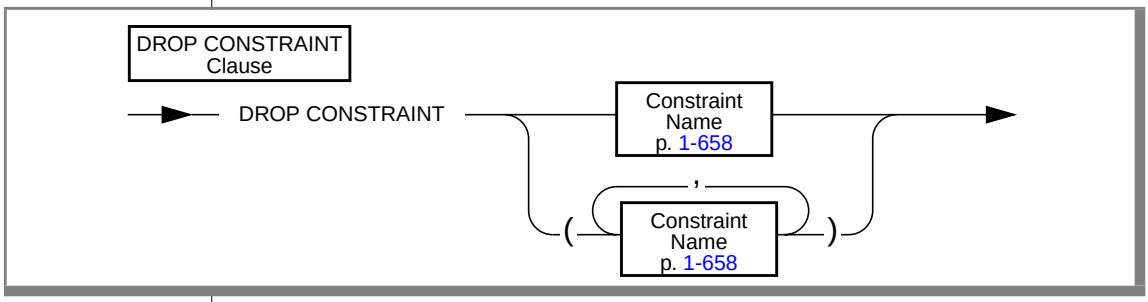
Privileges Required for Adding Constraints

When you own the table or have the Alter privilege on the table, you can create a unique, primary-key, or check constraint on the table and specify yourself as the owner of the constraint. To add a referential constraint, you must have the References privilege on either the referenced columns or the referenced table. When you have the DBA privilege, you can create constraints for other users.

Recovery from Constraint Violations

If you use the ADD CONSTRAINT clause to add a table-level constraint in the enabled mode and receive an error message because existing rows would violate the constraint, you can follow a procedure to add the constraint successfully. See "Adding a Constraint When Existing Rows Violate the Constraint" on page 1-68.

DROP CONSTRAINT Clause



Use the DROP CONSTRAINT clause to drop any type of constraint, including not null constraints.

To drop an existing constraint, specify the DROP CONSTRAINT keywords and the name of the constraint. The following statement is an example of dropping a constraint:

```
ALTER TABLE manufact DROP CONSTRAINT con_name
```

If a *constraint name* is not specified when the constraint is created, the database server generates the name. You can query the **sysconstraints** system catalog table for the names (including the owner) of constraints. For example, to find the name of the constraint placed on the **items** table, you can issue the following statement:

```
SELECT constrname FROM  sysconstraints
    WHERE tabid = (SELECT tabid FROM systables
        WHERE tabname = 'items')
```

When you drop a unique or primary-key constraint that has a corresponding foreign key, the referential constraints is dropped. For example, if you drop the primary-key constraint on the **order_num** column in the **orders** table and **order_num** exists in the **items** table as a foreign key, that referential relationship is also dropped.

MODIFY NEXT SIZE Clause

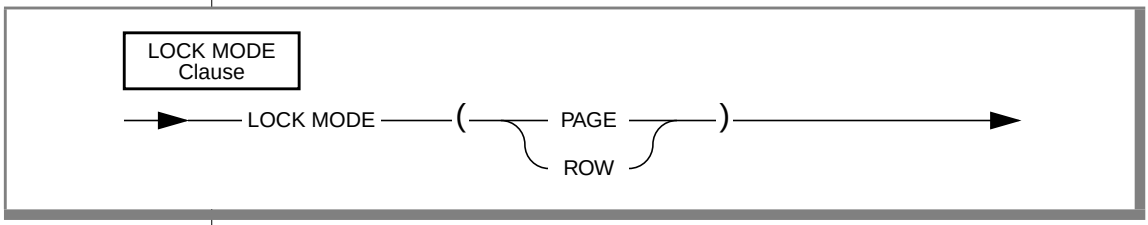
Element	Purpose	Restrictions	Syntax
kbytes	The length in kilobytes that you want to assign for the next extent for this table	The minimum length is four times the disk page size on your system. For example, if you have a 2-kilobyte page system, the minimum length is 8 kilobytes. The maximum length is equal to the chunk size.	Expression, p. 1-671

Use the MODIFY NEXT SIZE clause to change the size of new extents. If you want to specify an extent size of 32 kilobytes, use a statement such as the one in the following example:

```
ALTER TABLE customer MODIFY NEXT SIZE 32
```

The size of existing extents is not changed.

LOCK MODE Clause



Use the LOCK MODE keywords to change the locking mode of a table. The default lock mode is PAGE; it is set if the table is created without using the LOCK MODE clause. You must use the LOCK MODE clause to change from page to row locking, as shown in the following example:

```
ALTER TABLE items LOCK MODE (ROW)
```

ADD ROWIDS Clause

Tip: *Use the ADD ROWIDS clause only on fragmented tables. In nonfragmented tables, the rowid column remains unchanged. Informix recommends that you use primary keys as an access method rather than exploiting the rowid column.*

By default, fragmented tables do not contain the "hidden" rowid column. You use the ADD ROWIDS clause to add a new column called **rowid** for use with fragmented tables. OnLine assigns a unique number to each row that remains stable for the life of the row. The database server creates an index that it uses when search to find the physical location of the row. The ADD ROWIDS clause cannot be used with other ALTER TABLE commands. After you add the rowid column, each row contains an additional 4 bytes to store the rowid value.

For additional information about the rowid column, refer to the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

DROP ROWIDS Clause

Use the DROP ROWIDS clause to drop a rowid column only if you created it with the CREATE TABLE or ALTER FRAGMENT statements on fragmented tables. You cannot drop the rowid columns of a nonfragmented table. The DROP ROWIDS clause cannot be used with any other ALTER TABLE commands.

References

See the CREATE TABLE, DROP TABLE, and LOCK TABLE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of data-integrity constraints and the discussion of the ON DELETE CASCADE clause in Chapter 4. Also see the discussion of creating a database and tables in Chapter 9.

See the SET statement in this manual for information on object modes.

BEGIN WORK

Use the BEGIN WORK statement to start a transaction (a sequence of database operations that the COMMIT WORK or ROLLBACK WORK statement terminates).

Syntax

Usage

The following code fragment shows how you might place statements within a transaction:

```
BEGIN WORK
LOCK TABLE stock
UPDATE stock SET unit_price = unit_price * 1.10
    WHERE manu_code = 'KAR'
DELETE FROM stock WHERE description = 'baseball bat'
INSERT INTO manufact (manu_code, manu_name, lead_time)
    VALUES ('LYM', 'LYMAN', 14)
COMMIT WORK
```

Each row that an UPDATE, DELETE, or INSERT statement affects during a transaction is locked and remains locked throughout the transaction. A transaction that contains many such statements or that contains statements affecting many rows can exceed the limits that your operating system or the INFORMIX-OnLine Dynamic Server configuration imposes on the maximum number of simultaneous locks. If no other user is accessing the table, you can avoid locking limits and reduce locking overhead by locking the table with the LOCK TABLE statement after you begin the transaction. Like other locks, this table lock is released when the transaction terminates.

You can issue the BEGIN WORK statement only if a transaction is not in progress. If you issue a BEGIN WORK statement while you are in a transaction, the database server returns an error.

ESQL

If you use the BEGIN WORK statement within a routine called by a WHENEVER statement, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. These statements prevent the program from looping if the ROLLBACK WORK statement encounters an error or a warning. ♦

With ANSI-Compliant Databases

ANSI

The BEGIN WORK statement is not needed because transactions are implicit. A warning is generated if you use a BEGIN WORK statement immediately after one of the following statements:

- DATABASE
- COMMIT WORK
- CREATE DATABASE
- ROLLBACK WORK
- START DATABASE

An error is generated if you use a BEGIN WORK statement after any other statement. ♦

References

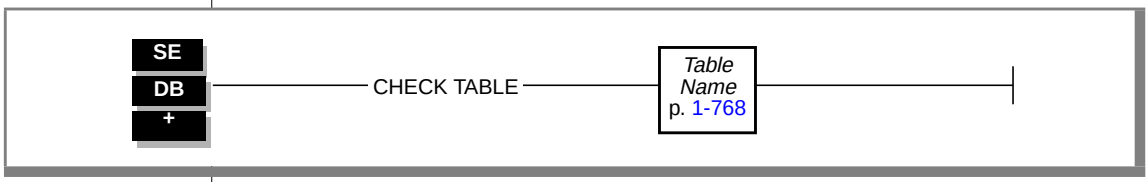
See the COMMIT WORK and ROLLBACK WORK statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of transactions and locking in Chapter 4 and Chapter 7, respectively.

CHECK TABLE

Use the CHECK TABLE statement to compare the data in a table with its indexes to determine whether they match. Use this statement when you think a power failure, computer crash, or other program interruption might have corrupted the data or the indexes. If the CHECK TABLE statement shows that a table is damaged, use the REPAIR TABLE statement to repair the table.

Syntax



Usage

Specify the name of the database table for which you want to check the data and associated indexes, as shown in the following example:

```
CHECK TABLE cust_calls
```

The CHECK TABLE statement calls the **secheck** utility. See the *INFORMIX-SE Administrator's Guide* for a full description of the **secheck** utility.

You must specify a table that is in a database on the current directory. If you specified a simple name for a database in the DATABASE command, but the database is not located in the current directory, CHECK TABLE does not search the **DBPATH** environment variable to find the directory for the database. The CHECK TABLE statement will fail. Similarly, if you specified an explicit pathname for a database in the DATABASE command, but the database is not located in the current directory, CHECK TABLE does not search for the database in the directory that you specified. The CHECK TABLE statement will fail.

You cannot use the CHECK TABLE statement on a table unless you own it or have the DBA privilege.

You cannot use the CHECK TABLE statement on the system catalog table **systables** because it is always open. Instead, you can run the **secheck** utility from the operating-system prompt. You cannot use the CHECK TABLE statement on other system catalog tables unless you are user **informix**.

References

See the REPAIR TABLE statement in this chapter.

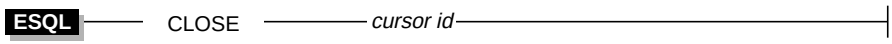
In the *INFORMIX-SE Administrator's Guide*, see the discussion of the **secheck** utility in Chapter 7.

CLOSE

Use the CLOSE statement when you no longer need to refer to the rows that a select or procedure cursor produced or when you want to flush and close an insert cursor.

Syntax

ESQL



Element	Purpose	Restrictions	Syntax
cursor id	The name of the cursor to be closed	The DECLARE statement must have previously declared the cursor.	Identifier, p. 1-723

Usage

Closing a cursor makes the cursor unusable for any statements except OPEN or FREE and releases resources that the database server had allocated to the cursor. A CLOSE statement treats a cursor that is associated with an INSERT statement differently than one that is associated with a SELECT or EXECUTE PROCEDURE statement.

You can close a cursor that was never opened or that has already been closed. No action is taken in these cases.

ANSI

You get an error if you close a cursor that was not open. No other action occurs. ♦

Closing a Select or Procedure Cursor

When *cursor id* is associated with a SELECT or EXECUTE PROCEDURE statement, closing the cursor terminates the SELECT or EXECUTE PROCEDURE statement. The database server releases all resources that it might have allocated to the active set of rows, for example, a temporary table that it used to hold an ordered set. The database server also releases any locks that it might have held on rows that were selected through the cursor. If a transaction contains the CLOSE statement, the database server does not release the locks until you execute COMMIT WORK or ROLLBACK WORK.

After you close a select or procedure cursor, you cannot execute a FETCH statement that names that cursor until you have reopened it.

Closing an Insert Cursor

When *cursor id* is associated with an INSERT statement, the CLOSE statement writes any remaining buffered rows into the database. The number of rows that were successfully inserted into the database is returned in the third element of the **sqlerrd** array in the **sqlca** structure (see the SQL API product-specific name in the following chart). For information on using SQLERRD to count the total number of rows that were inserted, see the PUT statement on page 1-416.

Product	Field Name
ESQL / C	sqlca.sqlerrd[2]
ESQL / COBOL	SQLERRD(3) OF SQLCA

The SQLCODE field of the **sqlca** structure indicates the result of the CLOSE statement for an insert cursor. If all buffered rows are successfully inserted, SQLCODE is set to zero. If an error is encountered, SQLCODE is set to a negative error message number. See the following chart for the field name for each SQL API product.

Product	Field Name
ESQL / C	sqlca.sqlcode
ESQL / COBOL	SQLCODE OF SQLCA

When SQLCODE is zero, the row buffer space is released, and the cursor is closed; that is, you cannot execute a PUT or FLUSH statement that names the cursor until you reopen it.

Tip: *When you encounter an SQLCODE error, a corresponding SQLSTATE error value might exist. Check the GET DIAGNOSTICS statement for information about how to get the SQLSTATE value and how to use the GET DIAGNOSTICS statement to interpret the SQLSTATE value.*

If the insert is not successful, the number of successfully inserted rows is stored in **sqlerrd**. Any buffered rows that follow the last successfully inserted row are discarded. Because the CLOSE statement failed in this case, the cursor is not closed. A second CLOSE statement can be successful because no buffered rows exist. A subsequent OPEN statement should also be successful because the OPEN statement performs a successful implicit close. For example, a CLOSE statement can fail if insufficient disk space prevents some of the rows from being inserted.

Using End of Transaction to Close a Cursor

The COMMIT WORK and ROLLBACK WORK statements close all cursors except those that are declared with hold. It is better to close all cursors explicitly, however. For select or procedure cursors, this action simply makes the intent of the program clear. It also helps to avoid a logic error if the WITH HOLD clause is later added to the declaration of a cursor.

For an insert cursor, it is important to use the CLOSE statement explicitly so that you can test the error code. Following the COMMIT WORK statement, SQLCODE reflects the result of the COMMIT statement, not the result of closing cursors. If you use a COMMIT WORK statement without first using a CLOSE statement, and if an error occurs while the last buffered rows are being written to the database, the transaction is still committed.

For the use of insert cursors and the WITH HOLD clause, see the DECLARE statement on page 1-234.

References

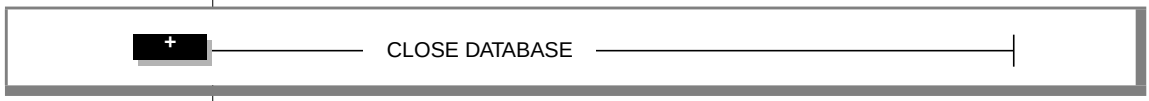
See the DECLARE, FETCH, FLUSH, FREE, OPEN, and PUT statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of cursors in Chapter 5.

CLOSE DATABASE

Use the CLOSE DATABASE statement to close the current database.

Syntax



Usage

Following the CLOSE DATABASE statement, the only legal SQL statements are CREATE DATABASE, DATABASE, and DROP DATABASE. A DISCONNECT statement can also follow a CLOSE DATABASE statement, but only if an explicit connection existed before you issue the CLOSE DATABASE statement. A CONNECT statement can follow a CLOSE DATABASE statement without any restrictions.

SE

You can also use the START DATABASE and ROLLFORWARD DATABASE statements after CLOSE DATABASE. ♦

Issue the CLOSE DATABASE statement before you drop the current database.

If your database has transactions, and if you have started a transaction, you must issue a COMMIT WORK statement before you use the CLOSE DATABASE statement.

The following example shows how to use the CLOSE DATABASE statement to drop the current database:

```
DATABASE stores7
.
.
.
CLOSE DATABASE
DROP DATABASE stores7
```

ESQL

The CLOSE DATABASE statement cannot appear in a multistatement PREPARE operation.

If you use the CLOSE DATABASE statement within a routine called by a WHENEVER statement, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. This action prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning.

When you issue the CLOSE DATABASE statement, declared cursors are no longer valid. You must redeclare any cursors that you want to use. ♦

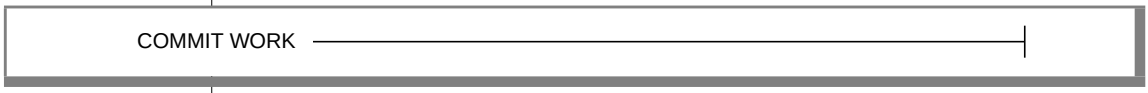
References

See the CONNECT, CREATE DATABASE, DATABASE, DISCONNECT, and DROP DATABASE statements in this manual.

COMMIT WORK

Use the COMMIT WORK statement to commit all modifications made to the database from the beginning of a transaction.

Syntax



Usage

Use the COMMIT WORK statement when you are sure you want to keep changes that are made to the database from the beginning of a transaction. Use the COMMIT WORK statement only at the end of a multistatement operation.

The COMMIT WORK statement releases all row and table locks.

ESQL

The COMMIT WORK statement closes all open cursors except those declared with hold. ♦

Issuing COMMIT WORK in a Database That Is Not ANSI Compliant

In a database that is not ANSI compliant, you must issue a COMMIT WORK statement at the end of a transaction if you initiated the transaction with a BEGIN WORK statement. If you fail to issue a COMMIT WORK statement in this case, the database server rolls back the modifications to the database that the transaction made.

If you are using a database that is not ANSI compliant, and you do not issue a BEGIN WORK statement, the database server executes each statement within its own transaction. These single-statement transactions do not require either a BEGIN WORK statement or a COMMIT WORK statement.

Issuing COMMIT WORK in an ANSI-Compliant Database

ANSI

In an ANSI-compliant database, you do not need to mark the beginning of a transaction. An implicit transaction is always in effect. You only need to mark the end of each transaction. A new transaction starts automatically after each COMMIT WORK or ROLLBACK WORK statement.

You must issue an explicit COMMIT WORK statement to mark the end of each transaction. If you fail to do so, the database server rolls back the modifications to the database that the transaction made. ♦

References

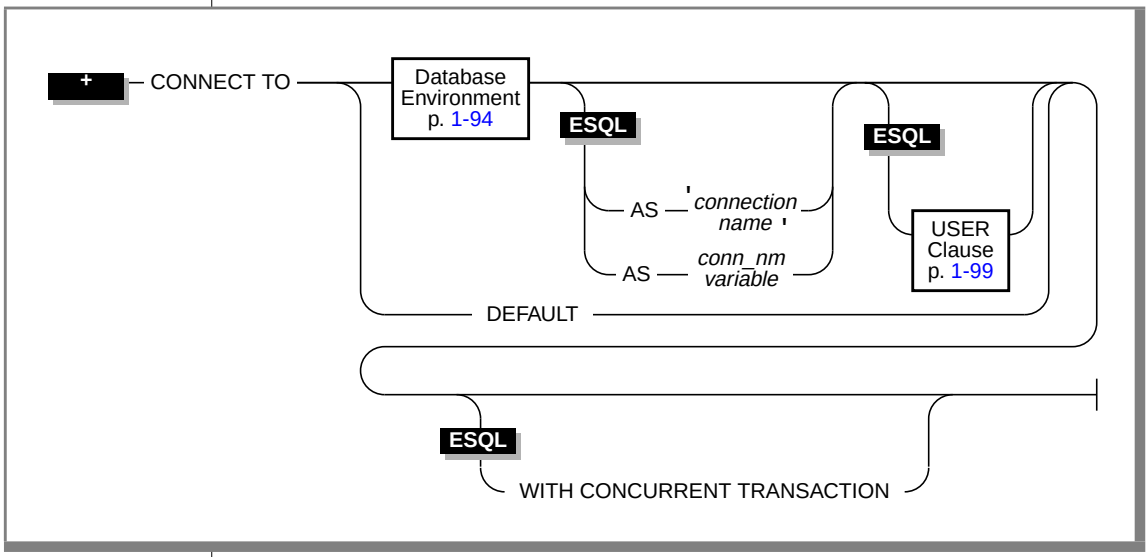
See the BEGIN WORK, ROLLBACK WORK, and DECLARE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of transactions in Chapter 4.

CONNECT

Use the CONNECT statement to connect to a database environment.

Syntax



Element	Purpose	Restrictions	Syntax
connection name	Quoted string that assigns a name to the connection	If your application makes multiple connections to the same database environment, you must specify a unique connection name for each connection.	Quoted String, p. 1-757
conn_nm variable	Host variable that holds the value of *connection name*	Variable must be a fixed-length character data type.	Variable name must conform to language-specific rules for variable names.

Usage

The CONNECT statement connects an application to a *database environment*. The database environment can be a database, a database server, or a database and a database server. If the application successfully connects to the specified database environment, the connection becomes the current connection for the application. SQL statements fail if no current connection exists between an application and a database server. If you specify a database name, the database server opens the database.You cannot use the CONNECT statement in a PREPARE statement.

An application can connect to several database environments at the same time, and it can establish multiple connections to the same database environment, provided each connection has a unique connection name. The only restriction on this is that an application can establish only one connection to each local server that uses the shared-memory connection mechanism. To find out whether a local server uses the shared memory connection mechanism or the local loopback connection mechanism, examine the **$INFORMIXDIR/etc/sqlhosts** file. (See the *INFORMIX-OnLine Dynamic Server Administrator's Guide* for more information.)

Only one connection is current at any time; other connections are dormant. The application cannot interact with a database through a dormant connection. When an application establishes a new connection, that connection becomes current, and the previous current transaction becomes dormant. You can make a dormant connection current with the SET CONNECTION statement. See "SET CONNECTION" on page 1-527.

Privileges for Executing the CONNECT Statement

The current user, or PUBLIC, must have the Connect database privilege on the database specified in the CONNECT statement.

The user who executes the CONNECT statement cannot have the same user name as an existing role in the database.

For information on using the USER clause to specify an alternate user name when the CONNECT statement connects to a database server on a remote host, see "USER Clause" on page 1-99.

Connection Identifiers

The optional *connection name* is a unique identifier that an application can use to refer to a connection in subsequent SET CONNECTION and DISCONNECT statements. If the application does not provide *connection name* (or a *conn_nm* host variable), it can refer to the connection using the database environment. If the application makes more than one connection to the same database environment, however, each connection must have a unique connection name.

After you associate a connection name with a connection, you can refer to the connection using only that connection name.

The value of *connection name* is case sensitive.

Connection Context

Each connection encompasses a set of information that is called the *connection context*. The connection context includes the name of the current user, the information that the database environment associates with this name, and information on the state of the connection (such as whether an active transaction is associated with the connection). The connection context is saved when an application becomes dormant, and this context is restored when the application becomes current again. (For more information on dormant connections, see "Making a Dormant Connection the Current Connection" on page 1-528.)

The DEFAULT Option

Use the DEFAULT option to request a connection to a default database server, called a *default connection*. The default database server can be either an INFORMIX-OnLine Dynamic Server or a INFORMIX-SE database server, and it can be local or remote. To designate the default database server, set its name in the environment variable **INFORMIXSERVER**. This form of the CONNECT statement does not open a database.

If you select the DEFAULT option for the CONNECT statement, you must use the DATABASE statement, the CREATE DATABASE statement, or the START DATABASE statement to open or create a database in the default database environment.

The Implicit Connection with DATABASE Statements

If you do not execute a CONNECT statement in your application, the first SQL statement must be one of the following database statements (or a single statement PREPARE for one of the following statements):

- DATABASE
- CREATE DATABASE
- START DATABASE
- DROP DATABASE

If one of these database statements is the first SQL statement in an application, the statement establishes a connection to a server, which is known as an *implicit* connection. If the database statement specifies only a database name, the database server name is obtained from the DBPATH environment variable. This situation is described in "Locating the Database" on page 1-97.

An application that makes an implicit connection can establish other connections explicitly (using the CONNECT statement) but cannot establish another implicit connection unless the original implicit connection is disconnected. An application can terminate an implicit connection using the DISCONNECT statement.

After *any* implicit connection is made, that connection is considered to be the default connection, regardless of whether the server is the default specified by the **INFORMIXSERVER** environment variable. This default allows the application to refer to the implicit connection if additional explicit connections are made, because the implicit connection does not have an identifier. For example, if you establish an implicit connection followed by an explicit connection, you can make the implicit connection current by issuing the SET CONNECTION DEFAULT statement. This means, however, that once you establish an implicit connection, you cannot use the CONNECT DEFAULT command because the implicit connection is considered to be the default connection.

The database statements can always be used to open a database or create a new database on the current database server.

The WITH CONCURRENT TRANSACTION Option

The WITH CONCURRENT TRANSACTION clause lets you switch to a different connection while a transaction is active in the current connection. If the current connection was *not* established using the WITH CONCURRENT TRANSACTION clause, you cannot switch to a different connection if a transaction is active; the CONNECT or SET CONNECTION statement fails, returning an error, and the transaction in the current connection continues to be active. In this case, the application must commit or roll back the active transaction in the current connection before it switches to a different connection.

The WITH CONCURRENT TRANSACTION clause supports the concept of multiple concurrent transactions, where each connection can have its own transaction and the COMMIT WORK and ROLLBACK WORK statements affect only the current connection.The WITH CONCURRENT TRANSACTION clause does not support global transactions in which a single transaction spans databases over multiple connections. The COMMIT WORK and ROLLBACK WORK statements do not act on databases across multiple connections.

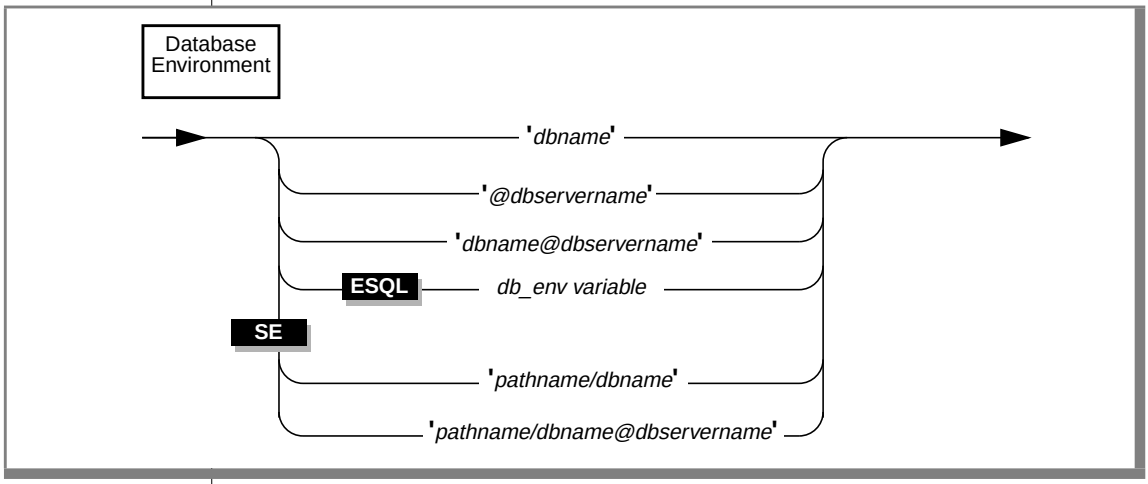
The following example illustrates how to use the WITH CONCURRRENT TRANSACTION clause:

```
main()
{
EXEC SQL connect to 'a@srv1' as 'A';
EXEC SQL connect to 'b@srv2' as 'B' with concurrent transaction;
EXEC SQL connect to 'c@srv3' as 'C' with concurrent transaction;

/*
    Execute SQL statements in connection 'C' , starting a
    transaction
*/

EXEC SQL set connection 'B'; -- switch to connection 'B'

/*
    Execute SQL statements starting a transaction in 'B'.
    Now there are two active transactions, one each in 'B'
    and 'C'.
*/

EXEC SQL set connection 'A'; -- switch to connection 'A'

/*
    Execute SQL statements starting a transaction in 'A'.
    Now there are three active transactions, one each in 'A',
    'B' and 'C'.
*/

EXEC SQL set connection 'C'; -- ERROR, transaction active in 'A'
```

```
/*
    SET CONNECTION 'C' fails (current connection is still 'A')
    The transaction in 'A' must be committed/rolled back since
    connection 'A' was started without the CONCURRENT TRANSACTION
    clause.
*/

EXEC SQL commit work;-- commit tx in current connection ('A')

/*
    Now, there are two active transactions, in 'B' and in 'C',
    which must be committed/rolled back separately
*/

EXEC SQL set connection 'B'; -- switch to connection 'B'
EXEC SQL commit work;        -- commit tx in current connection ('B')

EXEC SQL set connection 'C'; -- go back to connection 'C'
EXEC SQL commit work;      -- commit tx in current connection ('C')

EXEC SQL disconnect all;
}
```

Warning: *When an application uses the WITH CONCURRENT TRANSACTION clause to establish multiple connections to the same database environment, a deadlock condition can occur. A deadlock condition occurs when one transaction obtains a lock on a table, and a concurrent transaction tries to obtain a lock on the same table, resulting in the application waiting for itself to release the lock.*

Database Environment



Element	Purpose	Restrictions	Syntax
db_env variable	Host variable that contains a value representing a database environment	Variable must be a fixed-length character data type. The value stored in this host variable must have one of the database-environment formats listed in the syntax diagram.	Variable name must conform to language-specific rules for variable names.
dbname	Quoted string that identifies the name of the database to which a connection is made	Specified database must already exist. If you previously set the **DELIMIDENT** environment variable, surrounding quotes must be single. If the **DELIMIDENT** environment variable has not been previously set, surrounding quotes can be single or double.	Quoted String, p. 1-757
dbservername	Quoted string that identifies the name of the database server to which a connection is made	Specified database server must match the name of a server in the **sqlhosts** file. If you previously set the **DELIMIDENT** environment variable, surrounding quotes must be single. If the **DELIMIDENT** environment variable has not been previously set, surrounding quotes can be single or double.	Quoted String, p. 1-757
dbname@ dbservername	Quoted string that identifies the name of the database and database server to which a connection is made	Specified database must already exist. Specified database server must match the name of a server in the **sqlhosts** file. If you previously set the **DELIMIDENT** environment variable, surrounding quotes must be single. If the **DELIMIDENT** environment variable has not been previously set, surrounding quotes can be single or double.	Quoted String, p. 1-757

(1 of 2)

Element	Purpose	Restrictions	Syntax
pathname/ dbname	Quoted string that identifies the path of the database directory up to the parent directory of the **.dbs** directory (the directory where INFORMIX-SE database files reside). See "Locating the Database" on page 1-97 for the default actions taken by the database server if you omit the pathname.	Specified path must exist on the computer where the database server resides. Specified database must already exist.	Quoted String, p. 1-757. Pathname must conform to the rules of your operating system.
pathname/ dbname@ dbservername	Quoted string that identifies the path of the database directory up to the parent directory of the **.dbs** directory (the directory where INFORMIX-SE database files reside). The string also specifies the name of the INFORMIX-SE database server. See "Locating the Database" on page 1-97 for the default actions taken by the database server if you omit the pathname.	Specified path must exist on the computer where the database server resides. Specified database server must match the name of a server in the **sqlhosts** file. Specified database must already exist.	Quoted String p. 1-757. Pathname must conform to the rules of your operating system.

(2 of 2)

Specifying the Database Environment

Using the options shown in the syntax diagram, you can specify either a server and a database, a database server only, or a database only.

Specifying a Database Server Only

The *@dbservername* option establishes a connection to the named database server only; it does not open a database. When you use this option, you must subsequently use the DATABASE, CREATE DATABASE, or START DATABASE statement (or a PREPARE statement for one of these statements and an EXECUTE statement) to open a database.

Specifying a Database Only

The *dbname* option (for OnLine or SE) and the *pathname/dbname* option (for SE only) establish connections to the default server or to another database server in the DBPATH variable. It also locates and opens the named database.The same is true of the *db_env variable* option if it specifies only a database name. See "Locating the Database" below for the order in which an application connects to different database servers to locate a database.

Locating the Database

How a database is located and opened depends on both of the following factors:

- Whether you specify a database server name in the database environment expression
- Whether the database server is INFORMIX-OnLine Dynamic Server or INFORMIX-SE

Database Server and Database Specified

If you specify both a database server and a database in the CONNECT statement, your application connects to the database server, which locates and opens the database. If it is an OnLine database server, it uses parameters that are specified in the ONCONFIG configuration file to locate the database.

SE

The SE database server searches the directory that you supply. If you do not supply a directory path, it searches in the current directory (if the database server is local), the login directory (if the database server is remote), or the **DBPATH** environment variable. ♦

If the database server that you specify is not on-line, you get an error.

Only Database Specified

If you specify only a database in your CONNECT statement, not a database server, the application obtains the name of a database server from the **DBPATH** environment variable. The database server in the **INFORMIXSERVER** environment variable is always added in front of the **DBPATH** value specified by the user. Set environment variables as the following example shows:

```
setenv INFORMIXSERVER srvA
setenv DBPATH //srvB://srvC
```

The resulting **DBPATH** used by your application is shown in the following example:

```
//srvA://srvB://srvC
```

The application first establishes a connection to the database server specified by **INFORMIXSERVER**. If it is an OnLine database server, it uses parameters that are specified in the configuration file to locate the database.

If the database does not reside on the default database server, or if the default database server is not on-line, the application connects to the next database server in **DBPATH**. In the previous example, this server would be **srvB**.

SE

An SE database server searches the directory that you supply. If you do not supply a directory path, it searches in the current directory if the database server is local or the login directory if the database server is remote.

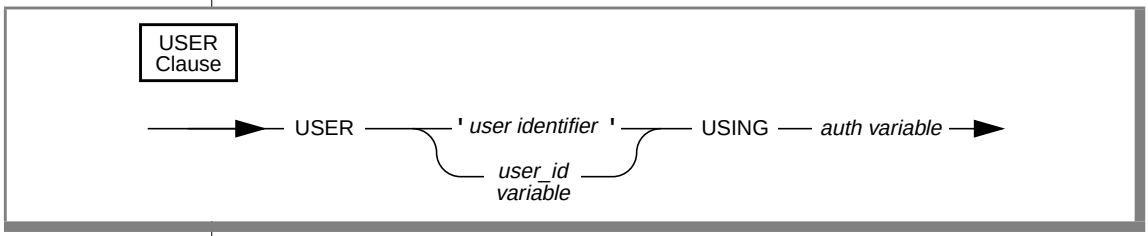
If a database server in **DBPATH** is an SE database server, it can contain a directory path. For example, the **DBPATH** might be as follows:

```
//srvB://srvC/usr/xyz
```

The database server will search for the database in the **/usr/xyz** directory. If an SE server in **DBPATH** does not have any directory path specified, the database server searches in the current directory if the database server is local or in the remote directory if the database server is remote. ♦

If a directory in **DBPATH** is an NFS-mounted directory, it is expanded to contain the host name of the NFS computer and the complete pathname of the directory on the NFS host. In this case, the host name must be listed in your **sqlhosts** file as a dbservername, and an **sqlexecd** daemon must be running on the NFS host.

USER Clause



Element	Purpose	Restrictions	Syntax
auth variable	Host variable that holds the valid password for the login name specified in *user identifier* or *user_id variable*	Variable must be a fixed-length character data type. The password stored in this variable must exist in the **/etc/passwd** file. If the application connects to a remote database server, the password must exist in this file on both the local and remote database servers.	Variable name must conform to language-specific rules for variable names.
user_id variable	The name of an ESQL/C or ESQL/COBOL host variable that holds the value of *user identifier*	Variable must be a fixed-length character data type. The login name stored in this variable is subject to the same restrictions as the *user identifier* variable.	Variable name must conform to language-specific rules for variable names.
user identifier	Quoted string that is a valid login name for the application	Specified login name must exist in the **/etc/passwd** file. If the application connects to a remote server, the login name must exist in this file on both the local and remote database servers.	Quoted String, p. 1-757

The User clause specifies information that is used to determine whether the application can access the target computer when the CONNECT statement connects to the database server on a remote host. Subsequent to the CONNECT statement, all database operations on the remote host use the specified user name.

The connection is rejected if the following conditions occur:

- The specified user lacks the privileges to access the database named in the database environment.
- The specified user does not have the required permissions to connect to the remote host.
- You supply a USER clause but do not include the USING *auth variable* phrase.

ESQL
X/O

In compliance with the X/Open specification for the CONNECT statement, the ESQL/C and ESQL/COBOL preprocessors allow a CONNECT statement that has a USER clause without the USING *auth variable* phrase. The connection is rejected at runtime by Informix database servers, however, if the *auth variable* is not present. ♦

If you do not supply the USER clause, the connection is attempted using the default user ID. The default Informix user ID is the login name of the user running the application. In this case, network permissions are obtained using the standard UNIX authorization procedures (for example, checking the **/etc/hosts.equiv** file).

Connecting to pre-6.0 INFORMIX-OnLine Dynamic Servers

The CONNECT statement syntax described in this chapter is valid for a Version 6.0 or later application connecting to pre-Version 6.0 database servers. As with Version 6.0 or later database servers, an implicit connection can be made to a pre-Version 6.0 server, provided that no existing implicit connections exist and no implicit connections have been previously terminated.

SE

You cannot connect to a pre-Version 6.0 server from a Version 6.0 or later application if the INFORMIX-SE database server has a nettype **seipcpip**. ♦

Connections to pre-Version 6.0 OnLine database server differ from connections to Version 6.0 and later database servers in the following respects:

- The CLOSE DATABASE statement causes a connection to a pre-Version 6.0 database server to be dropped. The same statement, applied to a connection to a Version 6.0 or later database server, causes the database to close, but the connection remains.
- If an application makes a connection to a pre-Version 6.0 database server without using the WITH CONCURRENT TRANSACTION clause, you must close the database (effectively dropping the connection) before switching to a different connection; otherwise, OnLine returns error -1800.

References

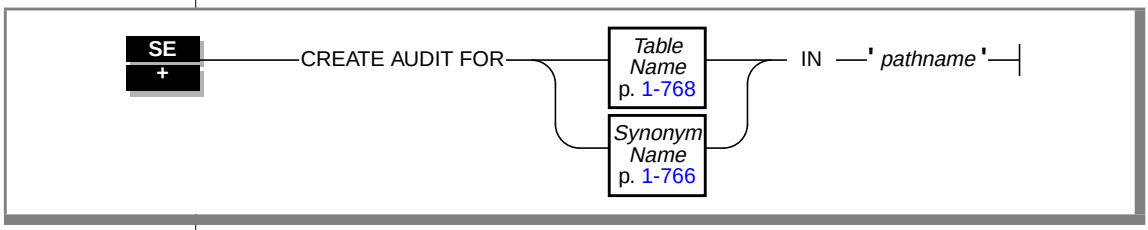
See the DISCONNECT, SET CONNECTION, DATABASE, START DATABASE, and CREATE DATABASE statements in this manual.

For information on the contents of the **sqlhosts** file, refer to the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

CREATE AUDIT

Use the CREATE AUDIT statement to create an audit-trail file and to start writing the audit trail for an INFORMIX-SE database.

Syntax



Element	Purpose	Restrictions	Syntax
pathname	The full operating-system pathname and filename for the audit-trail file	You can specify only one audit-trail file for a table. If an audit-trail file already exists for the table, the statement is not executed.	The pathname and filename must conform to the conventions of your operating system.

Usage

You can create an audit trail to keep a record of all modifications to a table. An audit trail is a complete history of all additions, deletions, and updates to the table. You can use the audit trail to reconstruct the table from a backup copy that is made when the audit trail is created.

You can use the CREATE AUDIT statement only with INFORMIX-SE. INFORMIX-OnLine Dynamic Server uses log files to provide full database logging.

To use the CREATE AUDIT statement, you must own the table or have the DBA privilege. You must set the Execute privilege for all directories below **root** in *pathname* for each class of user (owner, group, and public) that accesses your database.

If an audit-trail file with the same pathname already exists, the CREATE AUDIT statement does nothing. If an audit-trail file for the same table exists with a different pathname, an error message is returned.

Make a backup copy of your database files as soon as you run the CREATE AUDIT statement but before you make any further changes to the database. (See the RECOVER TABLE statement for an example.) If possible, put the audit trail file on a different physical device from the one that holds your data so that a failure of one does not damage the data on the other.

Audit trails slow your access to the database very slightly; each alteration of the database is recorded in the audit trail file as well as in the database files.

The following example shows how to use the CREATE AUDIT statement in a UNIX environment:

```
CREATE AUDIT FOR orders IN '/u/safe/orders.aud'
```

References

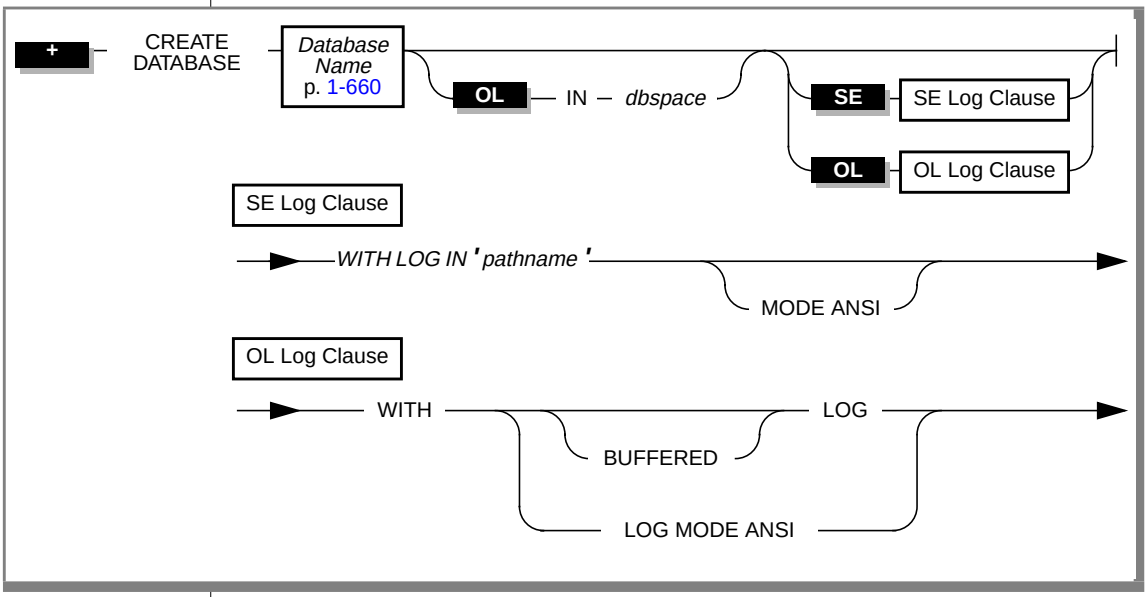
See the DROP AUDIT and RECOVER TABLE statements in this manual.

For more information on audit trails, see the *INFORMIX-SE Administrator's Guide*.

CREATE DATABASE

Use the CREATE DATABASE statement to create a new database.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace where you want to store the data for this database; default is the root dbspace	The dbspace must already exist.	Identifier, p. 1-723
pathname	The full pathname, including the file name, for the log file	You cannot specify an existing file.	The pathname and filename must conform to the conventions of your operating system.

Usage

The database that you create becomes the current database.

The database name that you use must be unique within the INFORMIX-OnLine Dynamic Server environment in which you are working. OnLine creates the system catalog tables that contain the data dictionary, which describes the structure of the database in the dbspace. If you do not specify the dbspace, OnLine creates the system catalog tables in the **root** dbspace.

When you create a database, you alone have access to it. The database remains inaccessible to other users until you, as DBA, grant database privileges. For information on granting database privileges, see the GRANT statement on page 1-340.

The following statement creates the **vehicles** database in the **root** dbspace:

```
CREATE DATABASE vehicles
```

The following statement creates the **vehicles** database in the **research** dbspace:

```
CREATE DATABASE vehicles IN research
```

SE

The following example creates the **vehicles** database in your current directory:

```
CREATE DATABASE vehicles
```

The data for the database is placed in a subdirectory of your current directory with the name *database-name*.dbs. The system catalog, tables, data, and index files are placed in this directory, except for tables that you explicitly instruct be placed elsewhere (see the CREATE TABLE statement on page 1-154). The rules for directory names on your operating system govern the length of the name that you choose for the database. ♦

ESQL

In SQL APIs, the CREATE DATABASE statement cannot appear in a multistatement PREPARE operation. ♦

ANSI-Compliant Databases

ANSI

You have the option of creating an ANSI-compliant database.

ANSI-compliant databases are set apart from databases that are not ANSI-compliant by the following features:

- All statements are automatically contained in transactions. All databases on the INFORMIX-OnLine Dynamic Server use unbuffered logging.
- Owner-naming is enforced. You must use the owner name when referring to each table, view, synonym, index, or constraint unless you are the owner.
- For databases on an OnLine database server, the default isolation level available is repeatable read.
- Default privileges on objects differ from those in databases that are not ANSI-compliant. Users do not receive PUBLIC privilege to tables and synonyms by default.

Other slight differences exist between databases that are ANSI-compliant and those that are not. These differences are noted as appropriate with the related <vk>SQL statement. ♦

Logging on INFORMIX-OnLine Dynamic Server

In the event of a failure, INFORMIX-OnLine Dynamic Server uses the log to re-create all committed transactions in your database.

If you do not specify the WITH LOG statement, you cannot use transactions or the statements that are associated with databases that have logging (BEGIN WORK, COMMIT WORK, ROLLBACK WORK, SET LOG, and SET ISOLATION).

Designating Buffered Logging

The following example creates a database that uses a buffered log:

```
CREATE DATABASE vehicles WITH BUFFERED LOG
```

If you use a buffered log, you marginally enhance the performance of logging at the risk of not being able to re-create the last few transactions after a failure. (See the discussion of buffered logging in Chapter 9 of the *Informix Guide to SQL: Tutorial*.)

ANSI

An ANSI-compliant database does not use buffered logging. ♦

Designating an ANSI-Compliant INFORMIX-OnLine Dynamic Server Database

The following example creates an ANSI-compliant database:

```
CREATE DATABASE employees WITH LOG MODE ANSI
```

Creating an ANSI-compliant database does not mean that you get ANSI warnings when you run the database. You must use the **-ansi** flag or the **DBANSIWARN** environment variable to receive warnings.

For additional information about **-ansi** and **DBANSIWARN**, see Chapter 4 in the *Informix Guide to SQL: Tutorial*.

Logging on INFORMIX-SE

SE

The following example creates an INFORMIX-SE database named **accounts** with a log file. You can use the full pathname and filename to designate the log file. If you specify a filename only, the log file is created in the current working directory.

```
CREATE DATABASE accounts WITH LOG IN '/acct/f1993/acct_log'
```

If you specify the WITH LOG IN keywords, you can use transactions and the statements that are associated with transactions (BEGIN WORK, COMMIT WORK, and ROLLBACK WORK). Conversely, if you do not specify the WITH LOG IN keywords, you cannot use transactions.

You can use the START DATABASE statement to assign a log file to an existing INFORMIX-SE database or to assign a new log file with a different name.

You can run the following SELECT statement to determine the location of the log file for the current database:

```
SELECT dirpath FROM informix.systables
    WHERE tabtype = 'L'
```

Designating an ANSI-Compliant INFORMIX-SE Database

The following example creates an ANSI-compliant database:

```
CREATE DATABASE employees WITH LOG IN '/u/acctg/lfile' MODE ANSI
```

♦

References

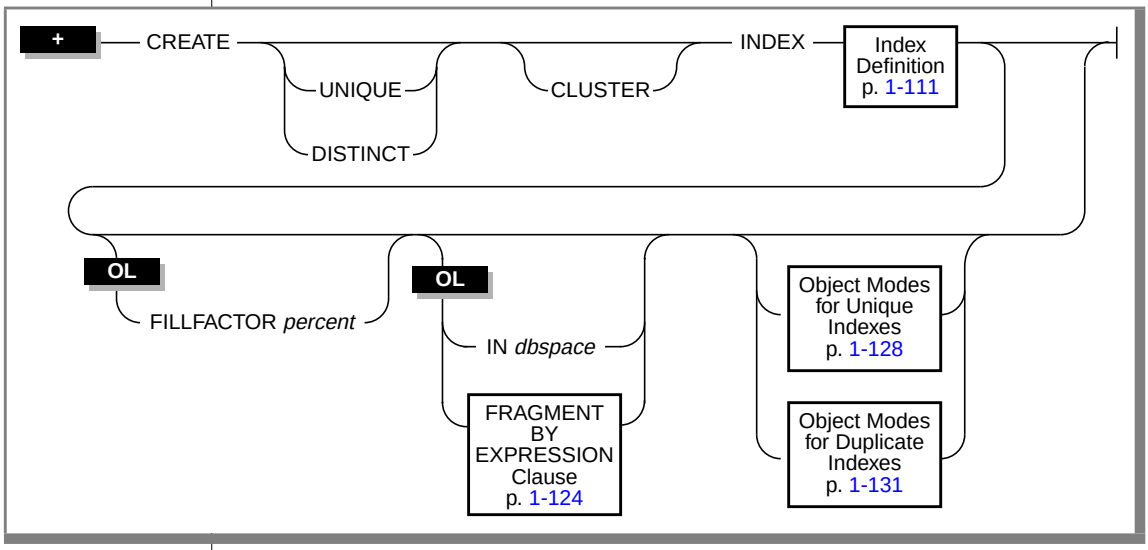
See the CLOSE DATABASE, CONNECT TO, DATABASE, DROP DATABASE, and START DATABASE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating a database in Chapter 9.

CREATE INDEX

Use the CREATE INDEX statement to create an index for one or more columns in a table and, optionally, to cluster the physical table in the order of the index. When more than one column is listed, the concatenation of the set of columns is treated as a single composite column for indexing. The indexes can be fragmented into separate dbspaces. You can create a unique or duplicate index, and you can set the object mode of either type of index.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace in which you want to place the index	The dbspace must exist at the time you execute the statement.	Identifier, p. 1-723
percent	The percentage of each index page that is filled by index data when the index is created. The default value is 90.	Value must be in the range 1 to 100.	Literal Number, p. 1-752

Usage

When you issue the CREATE INDEX statement, the table is locked in exclusive mode. If another process is using the table, the database server cannot execute the CREATE INDEX statement and returns an error.

SE

You cannot use a ROLLBACK WORK statement to undo a CREATE INDEX statement. If you roll back a transaction that contains a CREATE INDEX statement, the index remains, and you do not receive an error message. ♦

UNIQUE Option

The following example creates a unique index:

```
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num)
```

A unique index prevents duplicates in the **customer_num** column. A column with a unique index can have, at most, one null value. The DISTINCT keyword is a synonym for the keyword UNIQUE, so the following statement would accomplish the same task:

```
CREATE DISTINCT INDEX c_num_ix ON customer (customer_num)
```

The index in either example is maintained in ascending order, which is the default order.

If you do not specify the UNIQUE or DISTINCT keywords in a CREATE INDEX statement, a duplicate index is created. A duplicate index allows duplicate values in the indexed column.

You can also prevent duplicates in a column or set of columns by creating a unique constraint with the CREATE TABLE or ALTER TABLE statement. See the CREATE TABLE or ALTER TABLE statements for more information on creating unique constraints.

How Unique and Referential Constraints Affect Indexes

Internal indexes are created for unique and referential constraints. If a unique or referential constraint is added after the table is created, the user-created indexes are used, if appropriate. An appropriate index is one that indexes the same columns that are used in the referential or unique constraint. If an appropriate index is not available, a nonfragmented index is created in the database dbspace.

CLUSTER Option

Use the CLUSTER option to reorder the physical table in the order designated by the index. The CREATE CLUSTER INDEX statement fails if a CLUSTER index already exists.

```
CREATE CLUSTER INDEX c_clust_ix ON customer (zipcode)
```

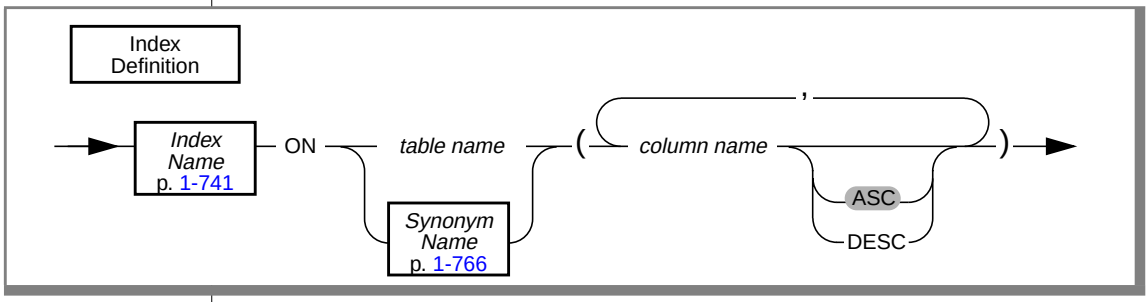
This statement creates an index on the **customer** table that orders the table physically by zip code.

SE

You cannot create a CLUSTER index on a table that has an audit trail. ♦

If the CLUSTER option is specified in addition to fragments on an index, the data is clustered only within the context of the fragment and not globally across the entire table.

Index Definition



Element	Purpose	Restrictions	Syntax
column name	The name of the column or columns that you want to index	You must observe restrictions on the location of the columns, the maximum number of columns, the total width of the columns, existing constraints on the columns, and the number of indexes allowed on the same columns. See "Restrictions on the Column Name Variable in CREATE INDEX" below.	Identifier, p. 1-723
table name	The name of the table on which the index is created	The table must exist. The table can be a regular database table or a temporary table.	Identifier, p. 1-723

Use the Index Definition portion of the CREATE INDEX statement to give a name to the index, specify the table on which the index is created, and specify the column or columns to be used for the index. In addition, the ASC and DESC keywords allow you to specify whether the index will be sorted in ascending or descending order.

Restrictions on the Column Name Variable in CREATE INDEX

Observe the following restrictions when you specify the *column name* variable:

- All the columns you specify must exist and must belong to the same table.
- The maximum number of columns and the total width of all columns vary with the database server. See "Composite Indexes" on page 1-113.
- You cannot add an ascending index to a column or column list that already has a unique constraint on it. See "The ASC and DESC Keywords" on page 1-113.
- The number of indexes you can create on the same column or same sequence of columns is restricted. See "Number of Indexes Allowed" on page 1-120.

Composite Indexes

The following example creates a composite index using the **stock_num** and **manu_code** columns of the **stock** table:

```
CREATE UNIQUE INDEX st_man_ix ON stock (stock_num, manu_code)
```

The index prevents any duplicates of a given combination of **stock_num** and **manu_code**. The index is in ascending order by default.

You can include 16 columns in a composite index. The total width of all indexed columns in a single CREATE INDEX statement cannot exceed 255 bytes.

SE

You can use up to eight columns in a composite index. The total width of all indexed columns in a single CREATE INDEX statement cannot exceed 120 bytes. ♦

Place columns in the composite index in the order from most frequently used to least frequently used.

The ASC and DESC Keywords

Use the ASC option to specify an index that is maintained in ascending order. The ASC option is the default ordering scheme. Use the DESC option to specify an index that is maintained in descending order. When a column or list of columns is defined as unique in a CREATE TABLE or ALTER TABLE statement, the database server implements that UNIQUE CONSTRAINT by creating a unique ascending index. Thus, you cannot use the CREATE INDEX statement to add an ascending index to a column or column list that is already defined as unique.

You can create a descending index on such columns, and you can include such columns in composite ascending indexes in different combinations. For example, the following sequence of statements is allowed:

```
CREATE TABLE customer (
    customer_num        SERIAL(101) UNIQUE,
    fname               CHAR(15),
    lname               CHAR(15),
    company             CHAR(20),
    address1            CHAR(20),
    address2            CHAR(20),
    city                CHAR(15),
    state               CHAR(2),
    zipcode             CHAR(5),
    phone               CHAR(18)
                    )

CREATE INDEX cathtmp ON customer (customer_num DESC)
CREATE INDEX c_temp2 ON customer (customer_num, zipcode)
```

Bidirectional Traversal of Indexes

When you create an index on a column but do not specify the ASC or DESC keywords, the database server stores the key values in ascending order by default. If you specify the ASC keyword, the database server stores the key values in ascending order. If you specify the DESC keyword, the database server stores the key values in descending order.

Ascending order means that the key values are stored in order from the smallest key to the largest key. For example, if you create an ascending index on the **lname** column of the **customer** table, last names are stored in the index in the following order: `Albertson, Beatty, Currie.`

Descending order means that the key values are stored in order from the largest key to the smallest key. For example, if you create a descending index on the **lname** column of the **customer** table, last names are stored in the index in the following order: `Currie, Beatty, Albertson.`

However, the bidirectional traversal capability of the database server lets you create just one index on a column and use that index for queries that specify sorting of results in either ascending or descending order of the sort column.

Example of Bidirectional Traversal of an Index

An example can help to illustrate the bidirectional traversal of indexes by the database server. Suppose that you want to enter the following two queries:

```
SELECT lname, fname FROM customer ORDER BY lname ASC;
SELECT lname, fname FROM customer ORDER BY lname DESC;
```

When you specify the ORDER BY clause in SELECT statements such as these, you can improve the performance of the queries by creating an index on the ORDER BY column. Because of the bidirectional traversal capability of the database server, you only need to create a single index on the **lname** column.

For example, you can create an ascending index on the **lname** column with the following statement:

```
CREATE INDEX lname_bothways ON customer (lname ASC)
```

The database server will use the ascending index **lname_bothways** to sort the results of the first query in ascending order and to sort the results of the second query in descending order.

In the first query, you want to sort the results in ascending order. So the database server traverses the pages of the **lname_bothways** index from left to right and retrieves key values from the smallest key to the largest key. The query result is as follows.

lname	fname
Albertson	Frank
Beatty	Lana
Currie	Philip
.	
.	
.	
Vector	Raymond
Wallack	Jason
Watson	George

Traversing the index from left to right means that the database server starts at the leftmost leaf node of the index and continues to the rightmost leaf node of the index. For an explanation of leaf nodes in indexes, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

In the second query, you want to sort the results in descending order. So the database server traverses the pages of the **lname_bothways** index from right to left and retrieves key values from the largest key to the smallest key. The query result is as follows.

lname	fname
Watson	George
Wallack	Jason
Vector	Raymond
.	
.	
.	
Currie	Philip
Beatty	Lana
Albertson	Frank

Traversing the index from right to left means that the database server starts at the rightmost leaf node of the index and continues to the leftmost leaf node of the index. For an explanation of leaf nodes in indexes, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

Choosing an Ascending or Descending Index

In the preceding example, you created an ascending index on the **lname** column of the **customer** table by specifying the ASC keyword in the CREATE INDEX statement. Then the database server used this index to sort the results of the first query in ascending order of **lname** values and to sort the results of the second query in descending order of **lname** values. However, you could have achieved exactly the same results if you had created the index as a descending index.

For example, the following statement creates a descending index that the database server can use to process both queries:

```
CREATE INDEX lname_bothways2 ON customer (lname DESC)
```

The resulting **lname_bothways2** index stores the key values of the **lname** column in descending order, from the largest key to the smallest key. When the database server processes the first query, it traverses the index from right to left to perform an ascending sort of the results. When the database server processes the second query, it traverses the index from left to right to perform a descending sort of the results.

So it does not matter whether you create a single-column index as an ascending or descending index. Whichever storage order you choose for an index, the database server can traverse that index in ascending or descending order when it processes queries.

Use of the ASC and DESC Keywords in Composite Indexes

If you want to place an index on a single column of a table, you do not need to specify the ASC or DESC keywords because the database server can traverse the index in either ascending or descending order. The database server will create the index in ascending order by default, but the server can traverse this index in either ascending or descending order when it uses the index in a query.

However, if you create a composite index on a table, the ASC and DESC keywords might be required. For example, if you want to enter a SELECT statement whose ORDER BY clause sorts on multiple columns and sorts each column in a different order, and you want to use an index for this query, you need to create a composite index that corresponds to the ORDER BY columns.

For example, suppose that you want to enter the following query:

```
SELECT stock_num, manu_code, description, unit_price
    FROM stock
    ORDER BY manu_code ASC, unit_price DESC
```

This query sorts first in ascending order by the value of the **manu_code** column and then in descending order by the value of the **unit_price** column. To use an index for this query, you need to issue a CREATE INDEX statement that corresponds to the requirements of the ORDER BY clause. For example, you can enter either of following statements to create the index:

```
CREATE INDEX stock_idx1 ON stock
    (manu_code ASC, unit_price DESC);

CREATE INDEX stock_idx2 ON stock
    (manu_code DESC, unit_price ASC);
```

Now, when you execute the query, the database server uses the index that you created (either **stock_idx1** or **stock_idx2**) to sort the query results in ascending order by the value of the **manu_code** column and then in descending order by the value of the **unit_price** column. If you created the **stock_idx1** index, the database server traverses the index from left to right when it executes the query. If you created the **stock_idx2** index, the database server traverses the index from right to left when it executes the query.

Regardless of which index you created, the query result is as follows.

stock_num	manu_code	description	unit_price
8	ANZ	volleyball	$840.00
205	ANZ	3 golf balls	$312.00
110	ANZ	helmet	$244.00
304	ANZ	watch	$170.00
301	ANZ	running shoes	$95.00
310	ANZ	kick board	$84.00
201	ANZ	golf shoes	$75.00
313	ANZ	swim cap	$60.00
6	ANZ	tennis ball	$48.00
9	ANZ	volleyball net	$20.00
5	ANZ	tennis racquet	$19.80
309	HRO	ear drops	$40.00
302	HRO	ice pack	$4.50
.			
.			
.			
113	SHM	18-spd, assmbld	$685.90
1	SMT	baseball gloves	$450.00
6	SMT	tennis ball	$36.00
5	SMT	tennis racquet	$25.00

The composite index that was used for this query (**stock_idx1** or **stock_idx2**) cannot be used for queries in which you specify the same sort direction for the two columns in the ORDER BY clause. For example, suppose that you want to enter the following queries:

```
SELECT stock_num, manu_code, description, unit_price
    FROM stock
    ORDER BY manu_code ASC, unit_price ASC;

SELECT stock_num, manu_code, description, unit_price
    FROM stock
    ORDER BY manu_code DESC, unit_price DESC;
```

If you want to use a composite index to improve the performance of these queries, you need to enter one of the following CREATE INDEX statements. You can use either one of the created indexes (**stock_idx3** or **stock_idx4**) to improve the performance of the preceding queries.

```
CREATE INDEX stock_idx3 ON stock
    (manu_code ASC, unit_price ASC);

CREATE INDEX stock_idx4 ON stock
    (manu_code DESC, unit_price DESC);
```

Number of Indexes Allowed

Restrictions exist on the number of indexes that you can create on the same column or the same sequence of columns.

Restrictions on the Number of Indexes on a Single Column

You can create only one ascending index and one descending index on a single column. For example, if you wanted to create all possible indexes on the **stock_num** column of the **stock** table, you could create the following indexes:

- The **stock_num_asc** index on the **stock_num** column in ascending order
- The **stock_num_desc** index on the **stock_num** column in descending order

Because of the bidirectional traversal capability of the database server, you do not need to create both indexes in practice. You only need to create one of the indexes. Both of these indexes would achieve exactly the same results for an ascending or descending sort on the **stock_num** column. For further information on the bidirectional traversal capability of the database server, see "Bidirectional Traversal of Indexes" on page 1-114.

Restrictions on the Number of Indexes on a Sequence of Columns

You can create multiple indexes on a sequence of columns, provided that each index has a unique combination of ascending and descending columns. For example, to create all possible indexes on the **stock_num** and **manu_code** columns of the **stock** table, you could create the following indexes:

- The **ix1** index on both columns in ascending order
- The **ix2** index on both columns in descending order
- The **ix3** index on **stock_num** in ascending order and on **manu_code** in descending order
- The **ix4** index on **stock_num** in descending order and on **manu_code** in ascending order

Because of the bidirectional-traversal capability of the database server, you do not need to create these four indexes in practice. You only need to create two indexes:

- The **ix1** and **ix2** indexes achieve exactly the same results for sorts in which the user specifies the same sort direction (ascending or descending) for both columns. Therefore, you only need to create one index of this pair.
- The **ix3** and **ix4** indexes achieve exactly the same results for sorts in which the user specifies different sort directions for the two columns (ascending on the first column and descending on the second column or vice versa). Therefore, you only need to create one index of this pair.

For further information on the bidirectional-traversal capability of the database server, see "Bidirectional Traversal of Indexes" on page 1-114.

FILLFACTOR Clause

Use the FILLFACTOR clause to provide for expansion of the index at a later date or to create compacted indexes. You provide a percent value ranging from 1 to 100, inclusive. The default percent value is 90.

When the index is created, OnLine initially fills only that percentage of the nodes specified with the FILLFACTOR value. If you provide a low percentage value, such as 50, you allow room for growth in your index. The nodes of the index initially fill to a certain percentage and contain space for inserts. The amount of available space depends on the number of keys in each page as well as the percentage value. For example, with a 50-percent FILLFACTOR value, the page would be half full and could accommodate doubling in growth. A low percentage value can result in faster inserts and can be used for indexes that you expect to grow.

If you provide a high percentage value, such as 99, your indexes are compacted, and any new index inserts result in splitting nodes. The maximum density is achieved with 100 percent. With a 100-percent FILLFACTOR value, the index has no room available for growth; any additions to the index result in splitting the nodes. A 99-percent FILLFACTOR value allows room for at least one insertion per node. A high percentage value can result in faster selects and can be used for indexes that you do not expect to grow or for mostly read-only indexes.

This option takes effect only when you build an index on a table that contains more than 5,000 rows and uses more than 100 table pages, when you create an index on a fragmented table, or when you create a fragmented index on a nonfragmented table. The FILLFACTOR can also be set as a parameter in the ONCONFIG file. The FILLFACTOR clause on the CREATE INDEX statement overrides the setting in the ONCONFIG file.

For more information about the ONCONFIG file and the parameters you can use with **ONCONFIG**, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

Attached and Detached Indexes

When you fragment a table and, at a later time, create an index for that table, the index uses the same fragmentation strategy as the table unless you specify otherwise with the FRAGMENT BY EXPRESSION clause or the IN *dbspace* clause. Any changes to the table fragmentation result in a corresponding change to the index fragmentation. *Attached indexes* are indexes created without a fragmentation strategy. Indexes are *detached indexes* when they are created with a fragmentation strategy or stored in separate dbspaces from the table.

For information on the IN *dbspace* clause, see "The IN dbspace Clause" below. For information on the FRAGMENT BY EXPRESSION clause, see "The FRAGMENT BY EXPRESSION Clause" on page 1-124.

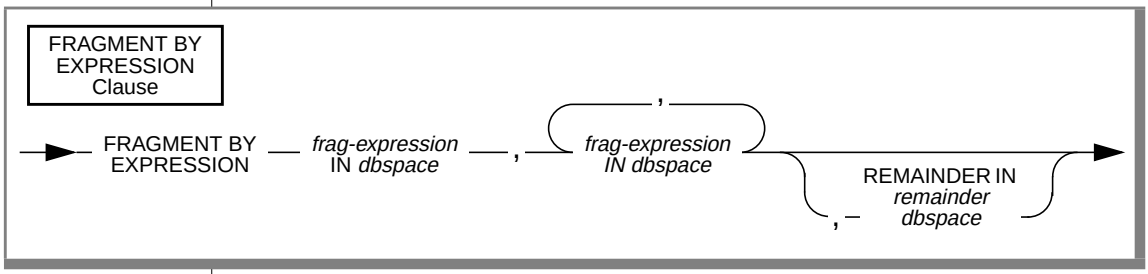
The IN dbspace Clause

Use the IN *dbspace* clause to specify the dbspace where you want your index to reside. With this clause, you create a detached index, even though the index is not fragmented. The dbspace that you specify must already exist. If you do not specify the IN *dbspace* clause, the index is created in the dbspace where the table was created. In addition, if you do not specify the IN *dbspace* clause, but the underlying table is fragmented, the index is created as a detached index, subject to all the restrictions on fragmented indexes. See "The FRAGMENT BY EXPRESSION Clause" on page 1-124 for more information about fragmented indexes.

The IN *dbspace* clause allows you to isolate an index. For example, if the **customer** table is created in the **custdata** dbspace, but you want to create an index in a separate dbspace called **custind**, use the following statements:

```
CREATE TABLE customer
    .
    .
    .
    IN custdata EXTENT SIZE 16

CREATE INDEX idx_cust ON customer (customer_num)
    IN custind
```

The FRAGMENT BY EXPRESSION Clause



Element	Purpose	Restrictions	Syntax
dbspace	The dbspace that will contain an index fragment that *frag-expression* defines	You must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces. The dbspaces must exist at the time you execute the statement.	Identifier, p. 1-723

(1 of 2)

Element	Purpose	Restrictions	Syntax
frag-expression	An expression that defines a fragment where an index key is to be stored using a range, hash, or arbitrary rule	If you specify a value for *remainder dbspace*, you must specify at least one fragment expression. If you do not specify a value for *remainder dbspace*, you must specify at least two fragment expressions. You can specify a maximum of 2,048 fragment expressions. Each fragment expression can contain only columns from the current table and only data values from a single row. The columns contained in a fragment expression must be the same as the indexed columns, or a subset of the indexed columns. No subqueries, stored procedures, current date/time functions, or aggregates are allowed in a fragment expression.	Expression, p. 1-671, and Condition, p. 1-643
remainder dbspace	The dbspace that contains index keys that do not meet the conditions defined in any fragment expression	If you specify two or more fragment expressions, *remainder dbspace* is optional. If you specify only one fragment expression, *remainder dbspace* is required. The dbspace specified in *remainder dbspace* must exist at the time you execute the statement.	Identifier, p. 1-723

(2 of 2)

You use the FRAGMENT BY EXPRESSION clause to define the expression-based distribution scheme.

In an *expression-based* distribution scheme, each fragment expression in a rule specifies a dbspace. Each fragment expression within the rule isolates data and aids the database server in searching for index keys. You can specify one of the following rules:

- Range rule

  A range rule specifies fragment expressions that use a range to specify which index keys are placed in a fragment, as the following example shows:

```
.
.
.
FRAGMENT BY EXPRESSION
c1 < 100 IN dbsp1,
c1 >= 100 and c1 < 200 IN dbsp2,
c1 >= 200 IN dbsp3;
```

- Hash rule

  A hash rule specifies fragment expressions that are created when you use a hash algorithm, which is often implemented with the MOD function, as the following example shows:

```
.
.
.
FRAGMENT BY EXPRESSION
MOD(id_num, 3) = 0 IN dbsp1,
MOD(id_num, 3) = 1 IN dbsp2,
MOD(id_num, 3) = 2 IN dbsp3;
```

- Arbitrary rule

  An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically includes the use of OR clauses to group data, as the following example shows:

```
.
.
.
FRAGMENT BY EXPRESSION
zip_num = 95228 OR zip_num = 95443 IN dbsp2,
zip_num = 91120 OR zip_num = 92310 IN dbsp4,
REMAINDER IN dbsp5;
```

Warning: *When you specify a date value in a fragment expression, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

Creating Index Fragments

When you fragment a table, all indexes for the table become fragmented the same as the table, unless you specify a different fragmentation strategy.

Fragmentation of Unique Indexes

You can fragment unique indexes only with a table that uses an expression-based distribution scheme. The columns referenced in the fragment expression must be part of the indexed columns. If your CREATE INDEX statement fails to meet either of these restrictions, the CREATE INDEX fails, and work is rolled back.

Fragmentation of System Indexes

System indexes (such as those used in referential constraints and unique constraints) utilize user indexes if they exist. If no user indexes can be utilized, system indexes remain nonfragmented and are moved to the dbspace where the database was created. To fragment a system index, create the fragmented index on the constraint columns, and then add the constraint using the ALTER TABLE statement.

Fragmentation of Indexes on Temporary Tables

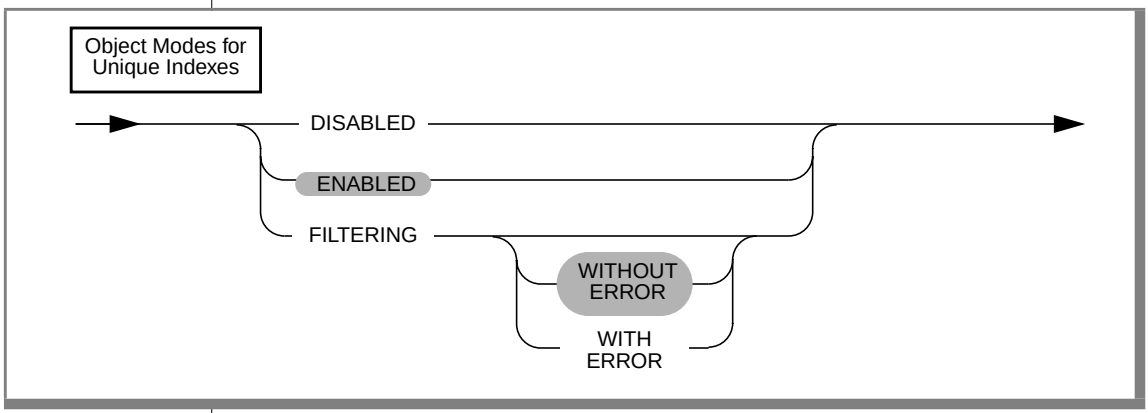
You can create explicit temporary tables with the TEMP TABLE clause of the CREATE TABLE statement or with the INTO TEMP clause of the SELECT statement. If you specified more than one dbspace in the **DBSPACETEMP** environment variable, but you did not specify an explicit fragmentation strategy, the database server fragments the temporary table round-robin across the dbspaces that **DBSPACETEMP** specifies.

If you then try to create a unique index on the temporary table, but you do not specify a fragmentation strategy for the index, the index is not fragmented in the same way as the table. You can fragment a unique index only if the underlying table uses an expression-based distribution scheme, but the temporary table is fragmented according to a round-robin distribution scheme.

Instead of fragmenting the unique index on the temporary table, the database server creates the index in the first dbspace that the **DBSPACETEMP** environment variable specifies. To avoid this result, use the FRAGMENT BY EXPRESSION clause to specify a fragmentation strategy for the index.

For more information on the **DBSPACETEMP** environment variable, see the *Informix Guide to SQL: Reference*.

Object Modes for Unique Indexes



You can set unique indexes in the following modes: disabled, enabled, and filtering. The following list explains these modes.

Object Mode	Effect
disabled	A unique index created in disabled mode is not updated after insert, delete, and update operations that modify the base table. Because the contents of the disabled index are not up to date, the optimizer does not use the index during the execution of queries.
enabled	A unique index created in enabled mode is updated after insert, delete, and update operations that modify the base table. Because the contents of the enabled index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a unique enabled index, the statement fails.
filtering	A unique index created in filtering mode is updated after insert, delete, and update operations that modify the base table. Because the contents of the filtering mode index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a unique index in filtering mode, the statement continues processing, but the bad row is written to the violations table associated with the base table. Diagnostic information about the unique-index violation is written to the diagnostics table associated with the base table.

If you specify filtering mode, you can also specify one of the following error options.

Error Option	Effect
WITHOUT ERROR	When a unique-index violation occurs during an insert or update operation, no integrity-violation error is returned to the user. You can specify this option only with the filtering-object mode.
WITH ERROR	When a unique-index violation occurs during an insert or update operation, an integrity-violation error is returned to the user. You can specify this option only with the filtering-object mode.

Specifying Object Modes for Unique Indexes

You must observe the following rules when you specify object modes for unique indexes in CREATE INDEX statements:

- You can set a unique index to the enabled, disabled, or filtering modes.
- If you do not specify the object mode of a unique index explicitly, the default mode is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option for a filtering-mode unique index, the default error option is WITHOUT ERROR.
- When you add a new unique index to an existing base table and specify the disabled object mode for the index, your CREATE INDEX statement succeeds even if duplicate values in the indexed column would cause a unique-index violation.
- When you add a new unique index to an existing base table and specify the enabled or filtering-object mode for the index, your CREATE INDEX statement succeeds provided that no duplicate values exist in the indexed column that would cause a unique-index violation. However, if any duplicate values exist in the indexed column, your CREATE INDEX statement fails and returns an error.
- When you add a new unique index to an existing base table in the enabled or filtering mode, and duplicate values exist in the indexed column, erroneous rows in the base table are not filtered to the violations table. Thus, you cannot use a violations table to detect the erroneous rows in the base table.

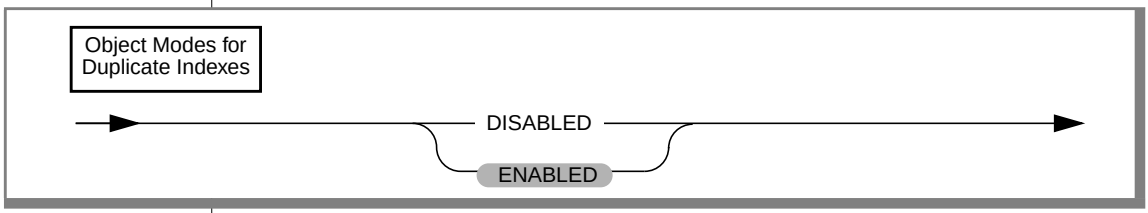
Adding a Unique Index When Duplicate Values Exist in the Column

If you attempt to add a unique index in the enabled mode but receive an error message because duplicate values are in the indexed column, take the following steps to add the index successfully:

1. Add the index in the disabled mode. Issue the CREATE INDEX statement again, but this time specify the DISABLED keyword.
2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.

3. Issue a SET statement to switch the object mode of the index to the enabled mode. When you issue this statement, existing rows in the target table that violate the unique-index requirement are duplicated in the violations table. However, you receive an integrity-violation error message, and the index remains disabled.
4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table. You might need to join the violations and diagnostics tables to get all the necessary information.
5. Take corrective action on the rows in the target table that violate the unique-index requirement.
6. After you fix all the nonconforming rows in the target table, issue the SET statement again to switch the disabled index to the enabled mode. This time the index is enabled, and no integrity violation error message is returned because all rows in the target table now satisfy the new unique-index requirement.

Object Modes for Duplicate Indexes



If you create a duplicate index, you can set the object mode of the index to the disabled or enabled mode. The following table explains these modes.

Object Mode	Effect
disabled	A duplicate index is created in disabled mode. The disabled index is not updated after insert, delete, and update operations that modify the base table. Because the contents of the disabled index are not up to date, the optimizer does not use the index during the execution of queries.
enabled	A duplicate index is created in enabled mode. The enabled index is updated after insert, delete, and update operations that modify the base table. Because the contents of the enabled index are up to date, the optimizer uses the index during the execution of queries. If an insert or update operation causes a duplicate key value to be added to a duplicate enabled index, the statement does not fail.

Specifying Object Modes for Duplicate Indexes

You must observe the following rules when you specify object modes for duplicate indexes in CREATE INDEX statements:

- You can set a duplicate index to the enabled or disabled mode, but you cannot set a duplicate index to the filtering mode.
- If you do not specify the object mode of a duplicate index explicitly, the default mode is enabled.

How the Database Server Treats Disabled Indexes

Whether a disabled index is a unique or duplicate index, the database server effectively ignores the index during data-manipulation operations.

When an index is disabled, the database server stops updating it and stops using it during queries, but the catalog information about the disabled index is retained. So you cannot create a new index on a column or set of columns if a disabled index on that column or set of columns already exists.

Similarly, you cannot create an active (not disabled) unique, foreign-key, or primary-key constraint on a column or set of columns if the indexes needed by the active constraint exist and are disabled.

References

See the ALTER INDEX, DROP INDEX, and CREATE TABLE statements in this manual.

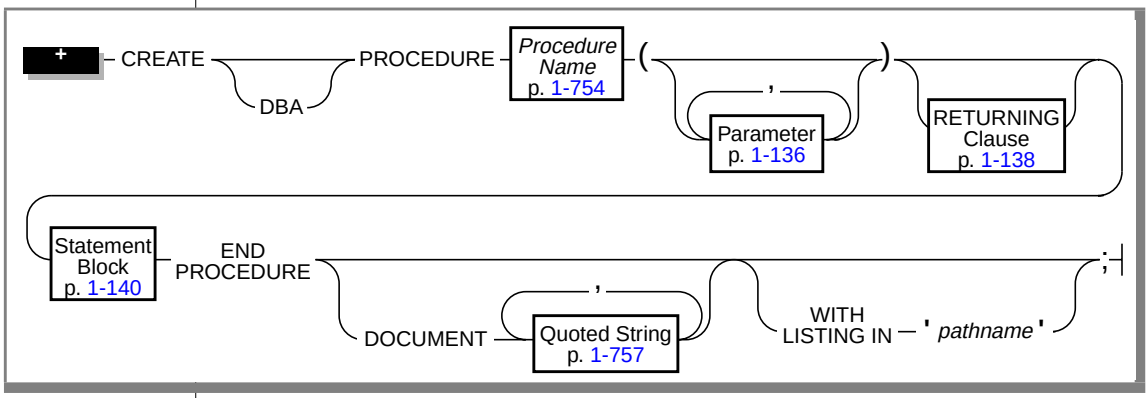
In the *Informix Guide to SQL: Tutorial*, see the discussions of indexes in Chapter 3. In addition, see the *INFORMIX-OnLine Dynamic Server Performance Guide* for information about performance issues with indexes.

In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the CREATE INDEX statement.

CREATE PROCEDURE

Use the CREATE PROCEDURE statement to name and define a stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
pathname	The pathname and filename of the listing file that is to contain warnings generated during the compilation of the procedure. See "WITH LISTING IN Option" on page 1-139 for the default action taken when you omit the pathname.	The specified pathname must exist on the computer where the database resides.	The pathname and filename must conform to the conventions of your operating system.

Usage

You must have the Resource privilege on a database to create a procedure.

The entire length of a CREATE PROCEDURE statement must be less than 64 kilobytes. This length is the literal length of the CREATE PROCEDURE statement, including blank space and tabs.

If the statement block portion of the CREATE PROCEDURE statement is empty, no operation takes place when you call the procedure. You might use such a procedure in the development stage when you want to establish the existence of a procedure but have not yet coded it.

Warning: *When you specify a date value in an expression in any statement in the statement block, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on how the database server interprets the date value. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect how the database server interprets the date value, so the stored procedure might produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

ESQL

You can use a CREATE PROCEDURE statement only within a PREPARE statement. If you want to create a procedure for which the text is known at compile time, you must use a CREATE PROCEDURE FROM statement. ♦

SE

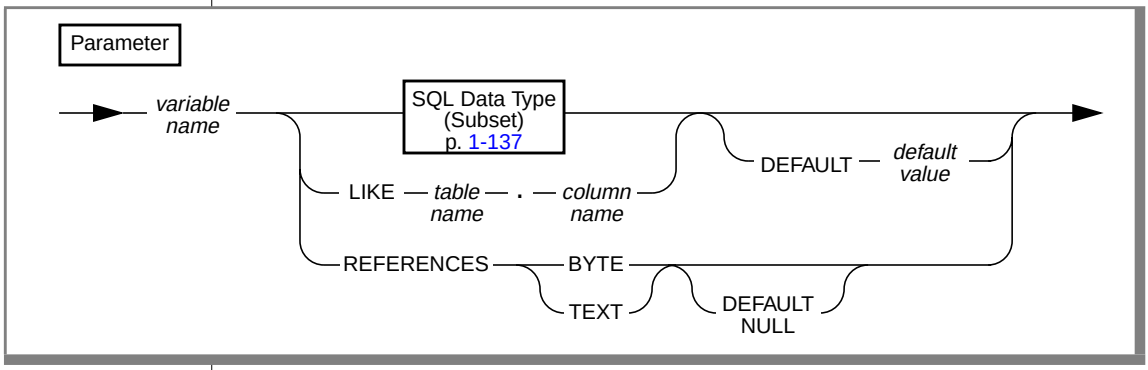
You cannot use a ROLLBACK WORK statement to undo a CREATE PROCEDURE statement. If you roll back a transaction that contains a CREATE PROCEDURE statement, the procedure remains, and you do not receive an error message. ♦

DBA Option

If you create a procedure using the DBA option, it is known as a DBA-privileged procedure. If you do not use the DBA option, the procedure is known as an owner-privileged procedure. The privileges associated with the execution of a procedure are determined by whether the procedure is created with the DBA keyword. See Chapter 12 of the *Informix Guide to SQL: Tutorial* for more information.

If you create an owner-privileged procedure in a database that is not ANSI-compliant, the **NODEFDAC** environment variable prevents privileges on that procedure from being granted to PUBLIC. See the *Informix Guide to SQL: Reference* for further information on the **NODEFDAC** environment variable.

Parameter Syntax and Use



Element	Purpose	Restrictions	Syntax
column name	The name of a column whose data type is assigned as *variable name*	The column must exist in the specified table.	Identifier, p. 1-723
default value	The default value that a procedure uses if you do not supply a value for *variable name* when you call the procedure	The default value must conform to the data type of *variable name*.	Expression, p. 1-671
table name	The name of the table that contains *column name*	The table must exist in the database.	Identifier, p. 1-723
variable name	The name of a parameter for which you supply a value when you call the procedure	You can specify zero, one, or more parameters in a CREATE PROCEDURE statement.	Identifier, p. 1-723

To define a parameter within the CREATE PROCEDURE statement, you must specify its name. You must also specify its data type. You can specify the data type directly or use the LIKE or REFERENCES clauses to identify the data type.

Specifying a Default Value for a Parameter

You can use the DEFAULT keyword followed by an expression to specify a default value for a parameter. If you provide a default value for a parameter, and the procedure is called with fewer arguments than were defined for that procedure, the default value is used. If you do not provide a default value for a parameter, and the procedure is called with fewer arguments than were defined for that procedure, the calling application receives an error.

The following example shows a CREATE PROCEDURE statement that specifies a default value for a parameter. This procedure finds the square of the *i* parameter. If the procedure is called without specifying the argument for the *i* parameter, the database server uses the default value 0 for the *i* parameter.

```
CREATE PROCEDURE square_w_default
    (i INT DEFAULT 0) {Specifies default value of i}
RETURNING INT; {Specifies return of INT value}
DEFINE j INT; {Defines procedure variable j}
LET j = i * i; {Finds square of i and assigns it to j}
RETURN j; {Returns value of j to calling module}
END PROCEDURE;
```

Warning: *When you specify a date value as the default value for a parameter, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on how the database server interprets the date value. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect how the database server interprets the date value, so the stored procedure might not use the default value that you intended. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

Subset of SQL Data Types Allowed in the Parameter List

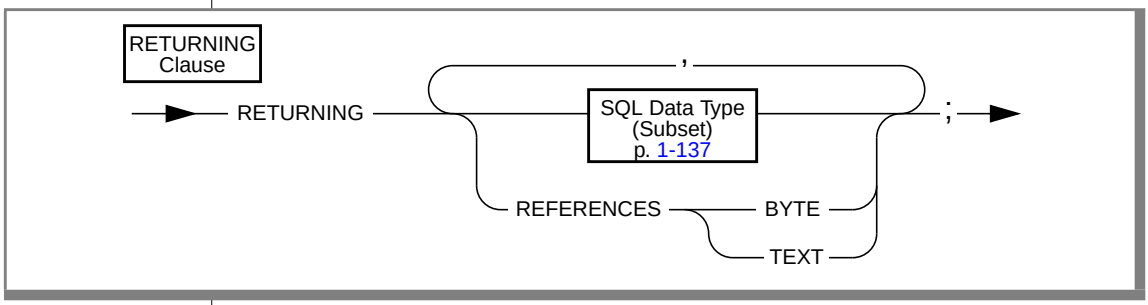
The SQL Data Type subset includes all the SQL data types except SERIAL, TEXT, and BYTE. For the complete syntax of all the SQL data types, see page 1-664.

To use a TEXT or BYTE data type, use the REFERENCES keyword, as the diagram on page 1-136 shows.

Referencing TEXT or BYTE Values

Use the REFERENCES clause to specify that a parameter contains TEXT or BYTE data. If you use the DEFAULT NULL option in the REFERENCES clause, and you call the procedure without a parameter, a null value is used.

RETURNING Clause



In the following examples, the RETURNING clause can be a noncursory procedure or a cursory procedure. In the first example, the RETURNING clause can return zero or one value if it is a noncursory procedure. However, if this clause is associated with a cursory procedure, it returns more than one row from a table, and each returned row contains zero or one value. In the second example, the RETURNING clause can return zero or two values if it is a noncursory procedure. However, if this clause is associated with a cursory procedure, it returns more than one row from a table and each returned row contains zero or two values.

```
RETURNING INT;

RETURNING INT, INT;
```

The receiving procedure or program must be written appropriately to accept the information.

Adding Comments to the Procedure

To add a comment to any line of a procedure, place a double-dash (--) before the comment or enclose the comment in curly brackets ({}). The double dash complies with the ANSI standard. The curly brackets are an Informix extension to the ANSI standard.

For examples of comments in procedures, see "Specifying a Default Value for a Parameter" on page 1-136. For detailed information on the double-dash (--) and curly-brackets ({}) symbols, see "How to Enter SQL Comments" on page 1-9.

Describing the Procedure in the DOCUMENT Clause

The quoted string or strings in the DOCUMENT clause provide a synopsis and description of the procedure. The DOCUMENT text is intended for the user of the procedure. Anyone with access to the database can query the **sysprocbody** system catalog table to obtain a description of one or all the procedures stored in the database.

For example, to find the description of the procedure called **do_something**, execute the following query:

```
SELECT data FROM sysprocbody b, sysprocedures p
WHERE b.procid   = p.procid {join between the two catalogs}
    AND
    p.procname = 'do_something' {look for procedure do_something}
    AND b.datakey  = 'D'   { want user document }
ORDER BY b.seqno;   {   ... in order     }
```

The rows returned are the complete text as supplied in the DOCUMENT clause of the CREATE PROCEDURE statement.

```
CREATE PROCEDURE ret_sal (dep_no INT, name CHAR(8) default user)
    RETURNING INT;
.
.
.
END PROCEDURE
DOCUMENT
    'Usage: salary (dept [required], name [default: your name])',
    'returns your (or someone else's) salary',
    'Warning: This procedure sends mail on illegal use'
WITH LISTING IN '/usr/tmp/sal.warnings';
```

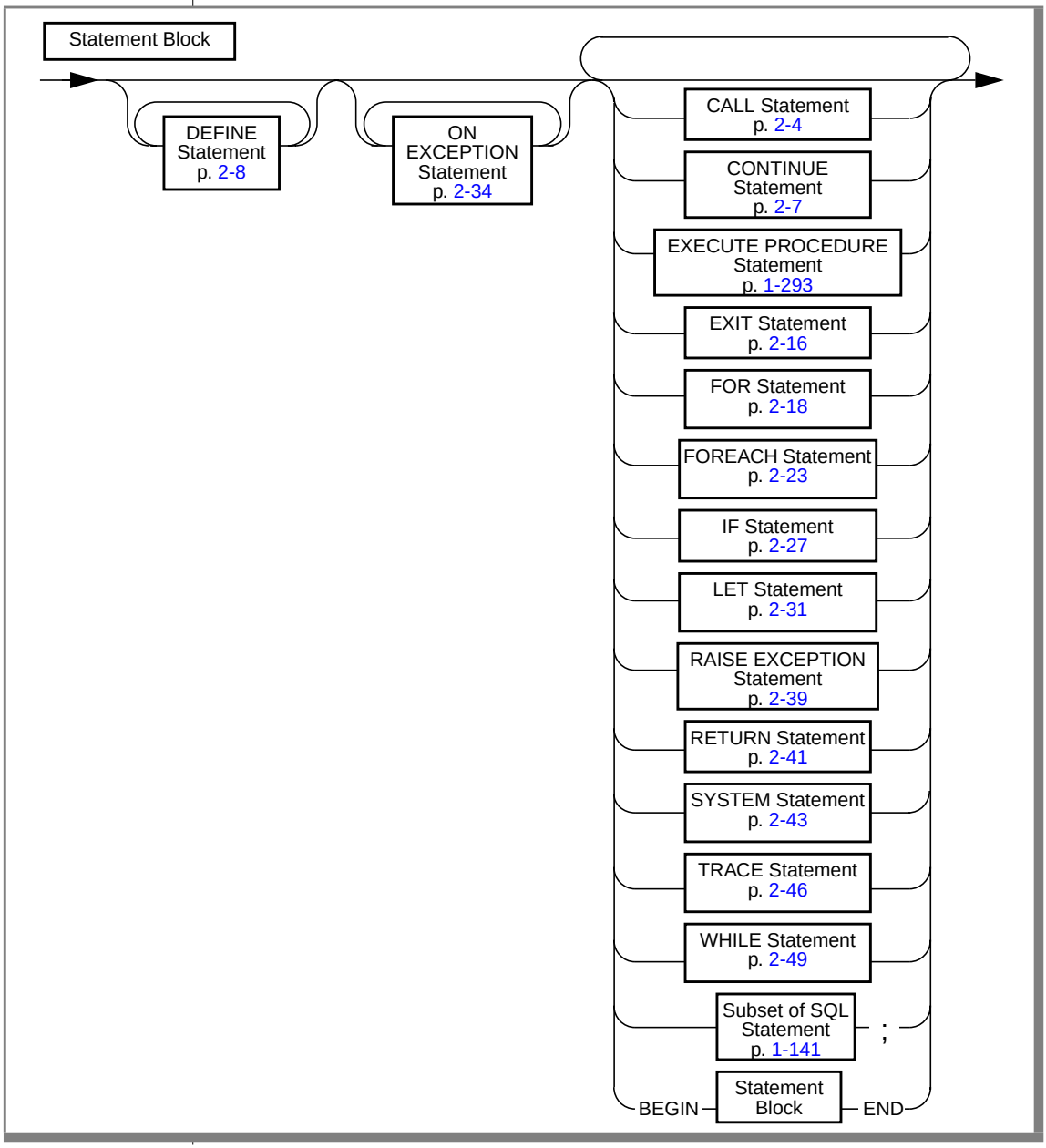
WITH LISTING IN Option

The WITH LISTING IN option specifies a filename where compile time warnings are sent. This listing file is created on the computer where the database resides. After you compile a procedure, this file holds one or more warning messages.

If you specify a filename but not a directory in the *pathname* variable, the default directory is your home directory on the computer where the database resides. If you do not have a home directory on this computer, the file is created in the root directory (the directory named "/").

If you do not use the WITH LISTING IN option, the compiler does not generate a list of warnings.

The Statement Block



The statement block can be empty, which results in a procedure that does nothing. Also, you cannot close the current database or select a new database within a procedure. And you cannot drop the current procedure within a procedure. You can, however, drop another procedure.

Subset of SQL Statements Allowed in the Statement Block

You can use any SQL statement in the statement block, except those listed in Figure 1-1.

Figure 1-1
SQL Statements That Cannot Be Used in a Stored Procedure

CHECK TABLE	FREE
CLOSE	GET DESCRIPTOR
CLOSE DATABASE	INFO
CONNECT	LOAD
CREATE DATABASE	OPEN
CREATE PROCEDURE	OUTPUT
CREATE PROCEDURE FROM	PREPARE
DATABASE	PUT
DECLARE	REPAIR
DESCRIBE	ROLLFORWARD DATABASE
EXECUTE	SET DESCRIPTOR
EXECUTE IMMEDIATE	START DATABASE
FETCH	UNLOAD
FLUSH	WHENEVER

Restrictions on SELECT Statement

You can use a SELECT statement in only two cases:

- You can use the INTO TEMP clause to put the results of the SELECT statement into a temporary table.
- You can use the SELECT...INTO form of the SELECT statement to put the resulting values into procedure variables.

Support for Roles and User Identity

You can use roles with stored procedures. You can execute role-related statements (CREATE ROLE, DROP ROLE, and SET ROLE) and SET SESSION AUTHORIZATION statements within a stored procedure. You can also grant privileges to roles with the GRANT statement within a procedure. Privileges that a user has acquired through enabling a role or by a SET SESSION AUTHORIZATION statement are not relinquished when a procedure is executed.

For further information about roles, see the CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE statements in this guide.

Restrictions on a Procedure Called in a Data Manipulation Statement

If a stored procedure is called as part of an INSERT, UPDATE, DELETE, or SELECT statement, the called procedure cannot execute any statement listed in Figure 1-2. This restriction ensures that the stored procedure cannot make changes that affect the SQL statement that contains the procedure call.

Figure 1-2
SQL Statements Not Allowed in a Procedure That a Data Manipulation Statement Calls

ALTER FRAGMENT	DROP SYNONYM
ALTER INDEX	DROP TABLE
ALTER OPTICAL	DROP TRIGGER
ALTER TABLE	DROP VIEW
BEGIN WORK	INSERT
COMMIT WORK	RENAME COLUMN
CREATE TRIGGER	RENAME TABLE
DELETE	ROLLBACK WORK
DROP DATABASE	SET CONSTRAINTS
DROP INDEX	UPDATE
DROP OPTICAL	

For example, if you use the following INSERT statement, the execution of the called procedure **dup_name** is restricted:

```
CREATE PROCEDURE sp_insert ()
.
.
.
INSERT INTO q_customer
    VALUES (SELECT * FROM customer
        WHERE dup_name ('lname') = 2)
.
.
.
END PROCEDURE;
```

In this example, **dup_name** cannot execute the statements listed in Figure 1-2. However, if **dup_name** is called within a statement that is not an INSERT, UPDATE, SELECT, or DELETE statement (namely EXECUTE PROCEDURE), **dup_name** can execute the statements listed in Figure 1-2.

You can use the BEGIN WORK and COMMIT WORK statements in procedures. You can start a transaction, finish a transaction, or start and finish a transaction in a procedure. If you start a transaction in a procedure that is executed remotely, you must finish the transaction before the procedure exits.

References

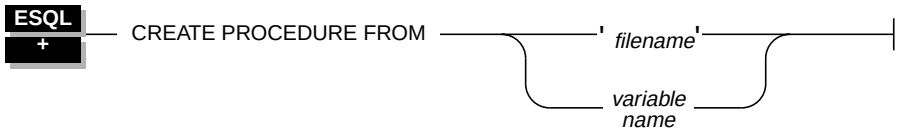
See the CREATE PROCEDURE FROM, DROP PROCEDURE, GRANT, EXECUTE PROCEDURE, PREPARE, UPDATE STATISTICS, and REVOKE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating and using stored procedures in Chapter 12.

CREATE PROCEDURE FROM

Use the CREATE PROCEDURE FROM statement to create a procedure. The actual text of the procedure resides in a separate file.

Syntax



Element	Purpose	Restrictions	Syntax
filename	The pathname and filename of the file that contains the full text of a CREATE PROCEDURE statement. The default pathname is the current directory.	The specified file must exist.	The pathname and filename must conform to the conventions of your operating system.
variable name	The name of a program variable that holds the value of *filename*	The file that is specified in the program variable must exist.	The name must conform to language-specific rules for variable names.

Usage

The filename that is provided in this statement is relative; if a simple filename is provided, the database server looks for the file in the current directory.

References

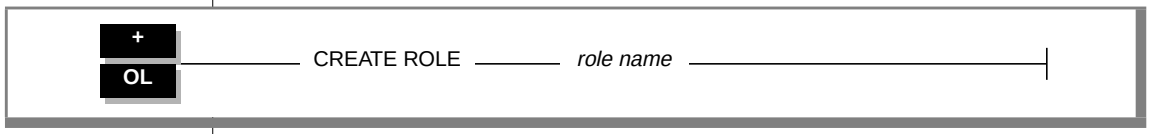
See the CREATE PROCEDURE statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating and using stored procedures in Chapter 12.

CREATE ROLE

Use the CREATE ROLE statement to create a new role.

Syntax



Element	Purpose	Restrictions	Syntax
role name	Name assigned to a role created by the DBA	Maximum number of characters is 8. A role name cannot be a user name known to the database server or the operating system of the database server. A role name cannot be in the **username** column of the **sysusers** system catalog table or in the **grantor** or **grantee** columns of the **systabauth**, **syscolauth**, **sysprocauth**, **sysfragauth**, and **sysroleauth** system catalog tables.	Identifier, p. 1-723

Usage

The database administrator (DBA) uses the CREATE ROLE statement to create a new role. A role can be considered as a classification, with privileges on database objects granted to the role. The DBA can assign the privileges of a related work task, such as **engineer**, to a role and then grant that role to users, instead of granting the same set of privileges to every user.

After a role is created, the DBA can use the GRANT statement to grant the role to users or to other roles. When a role is granted to a user, the user must use the SET ROLE statement to enable the role. Only then can the user use the privileges of the role.

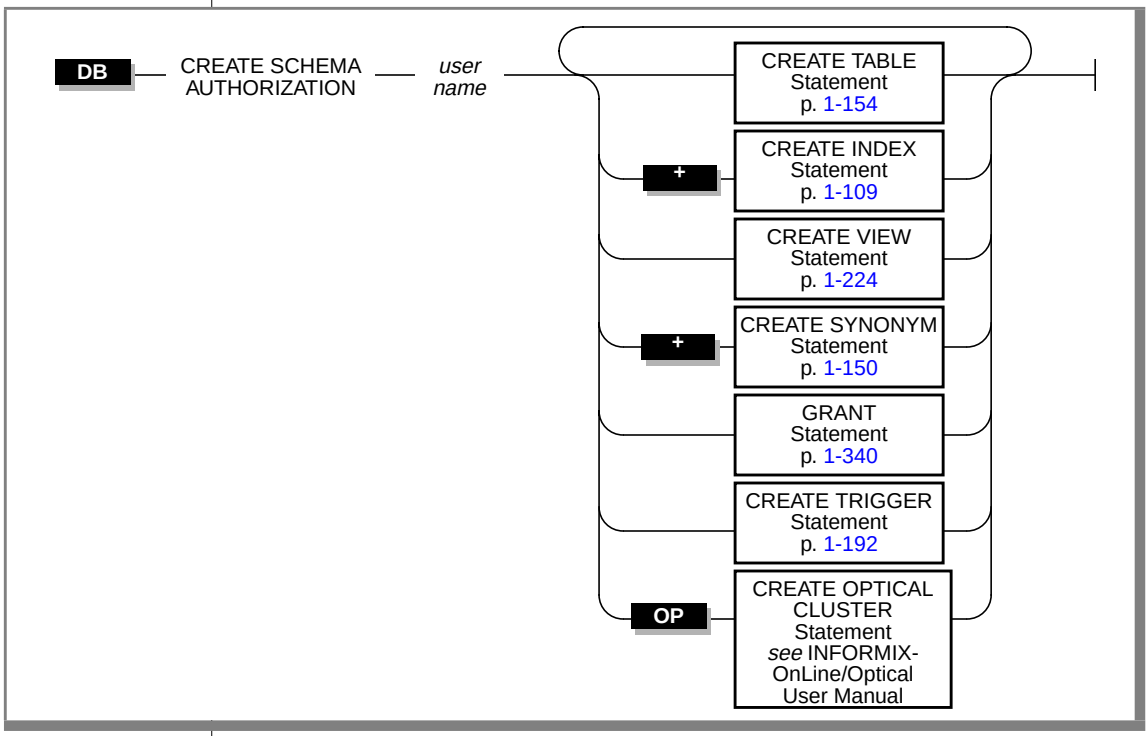
The CREATE ROLE statement, when used with the GRANT and SET ROLE statements, allows a DBA to create one set of privileges for a role and then grant the role to many use5 265steadt of(graningt thesamne set of privilegeston)]TJT*D0

CREATE SCHEMA

Purpose

The CREATE SCHEMA statement allows you to issue a block of CREATE and GRANT statements as a unit. It allows you to specify an owner of your choice for all objects that the CREATE SCHEMA statement creates.

Syntax



Element	Purpose	Restrictions	Syntax
user name	The name of the user who will own the objects that the CREATE SCHEMA statement creates	If the user who issues the CREATE SCHEMA statement has the Resource privilege, *user name* must be the name of this user. If the user who issues the CREATE SCHEMA statement has the DBA privilege, *user name* can be the name of this user or another user.	Identifier, p. 1-723

Usage

You cannot issue the CREATE SCHEMA statement until you create the affected database.

Users with the Resource privilege can create a schema for themselves. In this case, *user name* must be the name of the person with the Resource privilege who is running the CREATE SCHEMA statement. Anyone with the DBA privilege can also create a schema for someone else. In this case, *user name* can identify a user other than the person who is running the CREATE SCHEMA statement.

You can put CREATE and GRANT statements in any logical order within the statement, as the following example shows. Statements are considered part of the CREATE SCHEMA statement until a semicolon or an end-of-file symbol is reached.

```
CREATE SCHEMA AUTHORIZATION sarah
    CREATE TABLE mytable (mytime DATE, mytext TEXT)
    GRANT SELECT, UPDATE, DELETE ON mytable TO rick
    CREATE VIEW myview AS
        SELECT * FROM mytable WHERE mytime > '12/31/1993'
    CREATE INDEX idxtime ON mytable (mytime);
```

Creating Objects Within CREATE SCHEMA

All objects that a CREATE SCHEMA statement creates are owned by *user name*, even if you do not explicitly name each object. If you are the DBA, you can create objects for another user. If you are not the DBA, and you try to create an object for an owner other than yourself, you receive an error message.

Granting Privileges Within CREATE SCHEMA

You can only grant privileges with the CREATE SCHEMA statement; you cannot revoke or drop privileges.

Creating Objects or Granting Privileges Outside CREATE SCHEMA

If you create an object or use the GRANT statement outside a CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**.

References

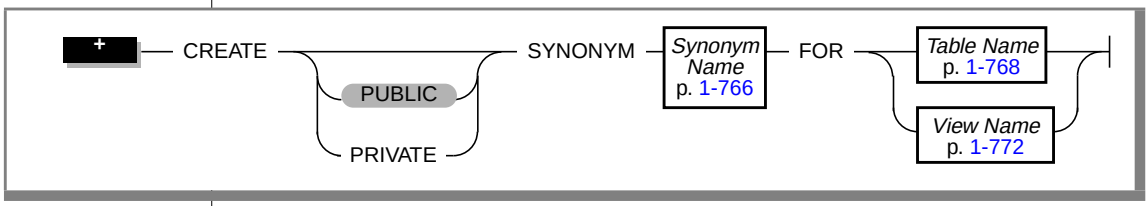
See the CREATE INDEX, CREATE SYNONYM, CREATE TABLE, CREATE VIEW, and GRANT statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating the database in Chapter 9.

CREATE SYNONYM

Use the CREATE SYNONYM statement to provide an alternative name for a table or view.

Syntax



Usage

Users have the same privileges for a synonym that they have for the table to which the synonym applies.

The synonym name must be unique; that is, the synonym name cannot be the same as another database object, such as a table, view, or temporary table.

Once a synonym is created, it persists until the owner executes the DROP SYNONYM statement. This property distinguishes a synonym from an alias that you can use in the FROM clause of a SELECT statement. The alias persists for the existence of the SELECT statement. If a synonym refers to a table or view in the same database, the synonym is automatically dropped if you drop the referenced table or view.

You cannot create a synonym for a synonym in the same database.

ANSI

The owner of the synonym (*owner.synonym*) qualifies the name of a synonym. The identifier *owner.synonym* must be unique among all the synonyms, tables, temporary tables, and views in the database. You must specify *owner* when you refer to a synonym that another user owns. The following example shows this convention:

```
CREATE SYNONYM emp FOR accting.employee
```

♦

SE

You cannot use a ROLLBACK WORK statement to undo a CREATE SYNONYM statement. If you roll back a transaction that contains a CREATE SYNONYM statement, the synonym remains, and you do not receive an error message. ♦

You can create a synonym for any table or view in any database on your database server. Use the *owner.* convention if the table is part of an ANSI-compliant database. The following example shows a synonym for a table outside the current database. It assumes that you are working on the same database server that contains the **payables** database.

```
CREATE SYNONYM mysum FOR payables:jean.summary
```

You can create a synonym for any table or view that exists on any networked database server as well as on the database server that contains your current database. The database server that holds the table must be on-line when you create the synonym. In a network, INFORMIX-OnLine Dynamic Server verifies that the object of the synonym exists when you create the synonym.

The following example shows how to create a synonym for an object that is not in the current database:

```
CREATE SYNONYM mysum FOR payables@phoenix:jean.summary
```

The identifier **mysum** now refers to the table **jean.summary**, which is in the **payables** database on the **phoenix** database server. Note that if the **summary** table is dropped from the **payables** database, the **mysum** synonym is left intact. Subsequent attempts to use **mysum** return the error `Table not found`.

PUBLIC and PRIVATE Synonyms

If you use the PUBLIC keyword (or no keyword at all), anyone who has access to the database can use your synonym. If a synonym is public, a user does not need to know the name of the owner of the synonym. Any synonym in a database that is not ANSI compliant *and* was created before Version 5.0 of the database server is a public synonym.

ANSI

Synonyms are always private. If you use the PUBLIC or PRIVATE keywords, you receive a syntax error. ♦

If you use the PRIVATE keyword, the synonym can be used only by the owner of the synonym or if the owner's name is specified explicitly with the synonym. More than one private synonym with the same name can exist in the same database. However, a different user must own each synonym with that name.

You can own only one synonym with a given name; you cannot create both private and public synonyms with the same name. For example, the following code generates an error:

```
CREATE SYNONYM our_custs FOR customer;
CREATE PRIVATE SYNONYM our_custs FOR cust_calls;-- ERROR!!!
```

Synonyms with the Same Name

If you own a private synonym, and a public synonym exists with the same name, when you use the synonym by its unqualified name, the private synonym is used.

If you use DROP SYNONYM with a synonym, and multiple synonyms exist with the same name, the private synonym is dropped. If you issue the DROP SYNONYM statement again, the public synonym is dropped.

Chaining Synonyms with INFORMIX-OnLine Dynamic Server

If you create a synonym for a table that is not in the current database, and this table is dropped, the synonym stays in place. You can create a new synonym for the dropped table, with the name of the dropped table as the synonym name, which points to another external or remote table. In this way, you can move a table to a new location and chain synonyms together so that the original synonyms remain valid. (You can chain as many as 16 synonyms in this manner.)

The following steps chain two synonyms together for the **customer** table, which will ultimately reside on the **zoo** database server (the CREATE TABLE statements are not complete):

1. In the **stores7** database on the database server that is called **training**, issue the following statement:

   ```
   CREATE TABLE customer (lname CHAR(15)...)
   ```

2. On the database server called **accntg**, issue the following statement:

   ```
   CREATE SYNONYM cust FOR stores7@training:customer
   ```

3. On the database server called **zoo**, issue the following statement:

```
CREATE TABLE customer (lname CHAR(15)...)
```

4. On the database server called **training**, issue the following statement:

```
DROP TABLE customer
CREATE SYNONYM customer FOR stores7@zoo:customer
```

The synonym **cust** on the **accntg** database server now points to the **customer** table on the **zoo** database server.

The following steps show an example of chaining two synonyms together and changing the table to which a synonym points:

1. On the database server called **training**, issue the following statement:

```
CREATE TABLE customer (lname CHAR(15)...)
```

2. On the database server called **accntg**, issue the following statement:

```
CREATE SYNONYM cust FOR stores7@training:customer
```

3. On the database server called **training**, issue the following statement:

```
DROP TABLE customer
CREATE TABLE customer (lastname CHAR(20)...)
```

The synonym **cust** on the **accntg** database server now points to a new version of the **customer** table on the **training** database server.

References

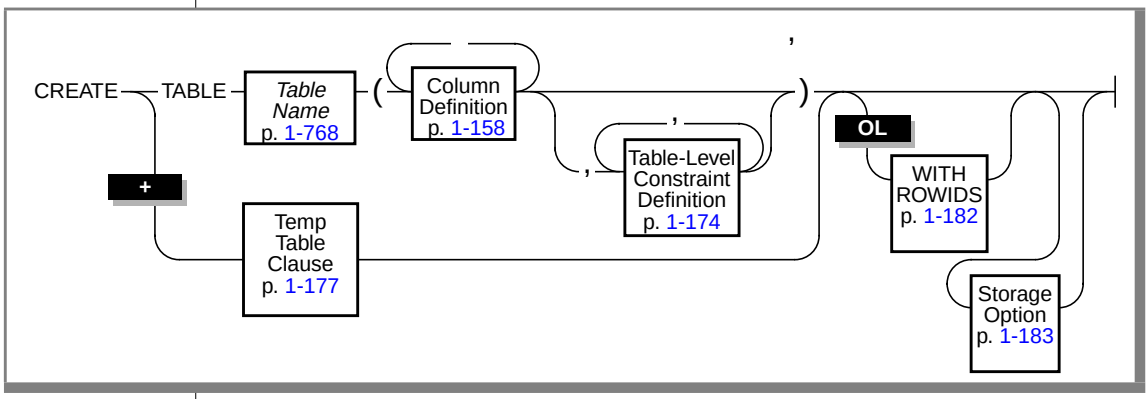
See the DROP SYNONYM statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of synonyms in Chapter 5 and Chapter 11

CREATE TABLE

Use the CREATE TABLE statement to create a new table in the current database, place data-integrity constraints on its columns or on a combination of its columns, designate the size of its initial and subsequent extents, and specify how to lock each table. The statement is also used to fragment tables into separate dbspaces.

Syntax



Usage

Table names must be unique within a database. However, although temporary table names must be different from existing table, view, or synonym names in the current database, they need not be different from other temporary table names used by other users.

ANSI

In an ANSI-compliant database, the combination *owner.tablename* must be unique within the database. ♦

SE

You cannot use a ROLLBACK WORK statement to undo a CREATE TABLE statement. If you roll back a transaction that contains a CREATE TABLE statement, the table remains, and you do not receive an error message. ♦

DB

Using the CREATE TABLE statement outside the CREATE SCHEMA statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

ESQL

Using the CREATE TABLE statement generates warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

Privileges on Tables

By default, all users who have been granted the Connect privilege to a database have all access privileges (except Alter and References) to the new table. To further restrict access, use the REVOKE statement to take *all* access away from PUBLIC (everyone on the system). Then, use the GRANT statement to designate the access privileges that you want to give to specific users.

When set to `yes`, the environment variable **NODEFDAC** prevents default privileges on a new table in a database that is not ANSI compliant from being granted to PUBLIC. For information about preventing privileges from being granted to PUBLIC, see the **NODEFDAC** environment variable in the *Informix Guide to SQL: Reference*.

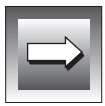
ANSI

In an ANSI-compliant database, no default table-level privileges exist. You must grant these privileges explicitly. ♦

Defining Constraints

When you create a table, several elements must be defined. For example, the table and columns within that table must have unique names. Also, every table column must have at least a data type associated with it. You can also, optionally, place several constraints on a given column. For example, you can indicate that the column has a specific default value or that data entered into the column must be checked to meet a specific data requirement.

Putting a constraint on a column is similar to putting an index on a column (using the CREATE INDEX statement). However, if you use constraints instead of indexes, you can also implement data-integrity constraints and turn effective checking off and on. For information on data-integrity constraints, refer to Chapter 3 of the *Informix Guide to SQL: Tutorial*. For information on effective checking, see the SET statement on page 1-501.



Important: *In a database without logging, detached checking is the only constraint-checking mode available. Detached checking means that constraint checking is performed on a row-by-row basis.*

You can define constraints at either the *column* or *table* level. If you choose to define constraints at the column level, you cannot have multiple-column constraints. In other words, the constraint created at the column level can apply only to a single column. If you choose to define constraints at the table level, you can apply constraints to single or multiple columns. At either level, single-column constraints are treated the same way.

Whenever you place a data restriction on a column, a constraint is created automatically. You have the option of specifying a name for the constraint. If you choose not to specify a name for the constraint, the database server creates a default constraint name for you automatically.

When you create a constraint of any type, the name of the constraint must be unique within the database.

ANSI

When you create a constraint of any type, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within the database. ♦

Limits on Constraint Definitions

You can include 16 columns in a list of columns for a unique, primary-key, or referential constraint. The total length of all columns cannot exceed 255 bytes.

SE

You can use up to eight columns, inclusive, in an INFORMIX-SE list of columns. The total length of all columns cannot exceed 120 bytes. ♦

You cannot place a constraint on a violations or diagnostics table. For further information on violations and diagnostics tables, see the START VIOLATIONS TABLE statement on page 1-584.

Adding or Dropping Constraints

After you have used the CREATE TABLE statement to place constraints on a column or set of columns, you must use the ALTER TABLE statement to add or drop the constraint from the column or composite column list.

Enforcing Primary-Key, Unique, and Referential Constraints

Primary-key, unique, and referential constraints are implemented as an ascending index that allows only unique entries or an ascending index that allows duplicates. When one of these constraints is placed on a column, the database server performs the following functions:

- Creates a unique index for a unique or primary-key constraint
- Creates a non-unique index for the columns specified in the referential constraint

However, if a constraint already was created on the same column or set of columns, another index is not built for the constraints. Instead, the existing index is *shared* by the constraints. If the existing index is non-unique, it is *upgraded* if a unique or primary-key constraint is placed on that column.

Because these constraints are enforced through indexes, you cannot create an index (using the CREATE INDEX statement) for a column that is of the same data type as the constraint placed on that column. For example, if a unique constraint exists on a column, you can create neither an ascending unique index for that column nor a duplicate ascending index.

Constraint Names

A row is added to the **sysindexes** system catalog table for each new primary-key, unique, or referential constraint that does not share an index with an existing constraint. The index name in the **sysindexes** system catalog table is created with the following format:

```
[space]<tabid>_<constraint id>
```

In this format, *tabid* and *constraint id* are from the **systables** and **sysconstraints** system catalog tables, respectively. For example, the index name might be something like this: " **121_13**" (quotes used to show the space).

The constraint name must be unique within the database. If you do not specify a *constraint name*, the database server generates one for the **sysconstraints** system catalog table using the following template:

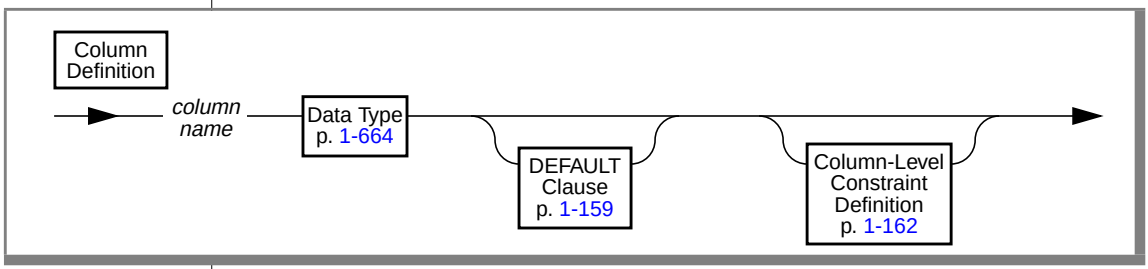
```
<constraint type><tabid>_<constraint id>
```

In this template, constraint type is the letter **u** for unique or primary-key constraints, **r** for referential constraints, **c** for check constraints, and **n** for not null constraints. For example, the constraint name for a unique constraint might look like this: **u111_14**. If the name conflicts with an existing identifier, the database server returns an error, and you must then supply a constraint name.

Object Modes for Constraints

You can specify the object mode for any type of column-level or table-level constraint. You can set the object mode of the constraint to the enabled, disabled, or filtering mode. See "Constraint-Mode Definitions" on page 1-163 for further information on object modes.

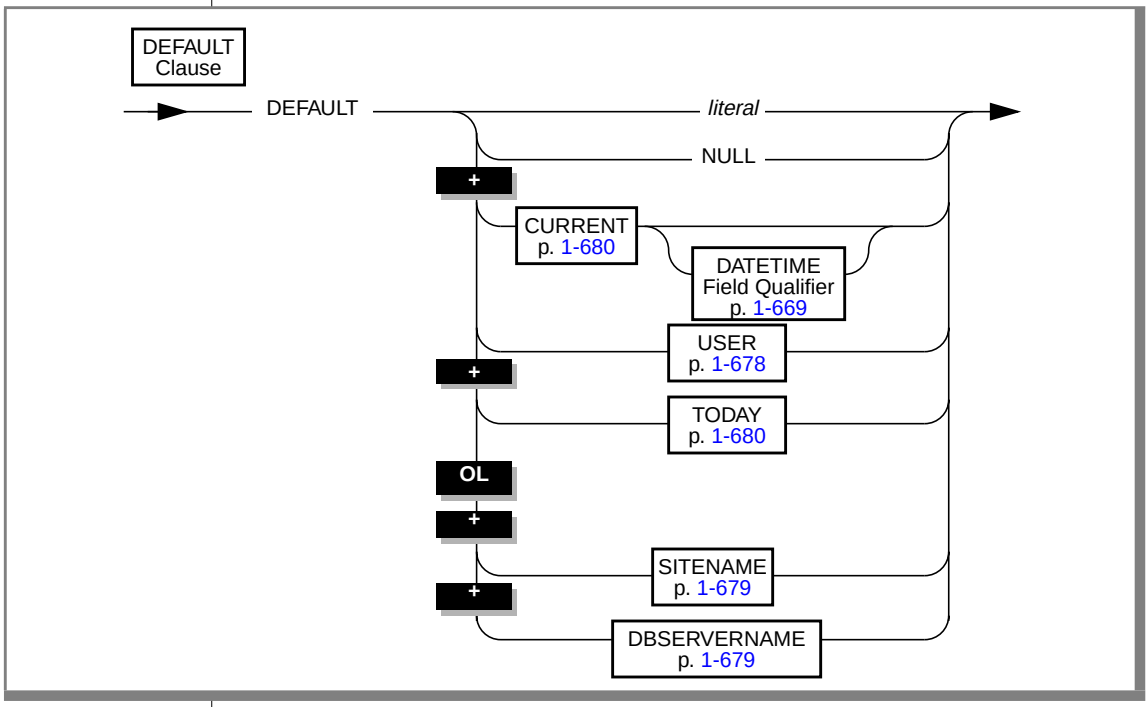
Column-Definition Option



Element	Purpose	Restrictions	Syntax
column name	The name of a column in the table	Name must be unique within a table, but you can use the same names in different tables in the same database.	Identifier, p. 1-723

Use the column-definition portion of the CREATE TABLE statement to list the name, data type, default values, and constraints *of a single column* as well as to indicate whether the column does not allow duplicate values.

The DEFAULT Clause



Element	Purpose	Restrictions	Syntax
literal	A literal term that defines alpha or numeric constant characters to be used as the default value for the column	Term must be appropriate type for the column. See "Literal Terms as Default Values" on page 1-160.	Expression, p. 1-671

The default value is inserted into the column when an explicit value is not specified. If a default is not specified, and the column allows nulls, the default is NULL. If you designate NULL as the default value for a column, you cannot specify a not null constraint as part of the column definition.

You cannot designate default values for serial columns. If the column is TEXT or BYTE data type, you can *only* designate NULL as the default value.

Literal Terms as Default Values

You can designate *literal terms* as default values. A literal term is a string of character or numeric constant characters that you define. To use a literal term as a default value, you must adhere to the following rules.

Use a Literal	With Columns of Data Type
INTEGER	INTEGER, SMALLINT, DECIMAL, MONEY, FLOAT, SMALLFLOAT
DECIMAL	DECIMAL, MONEY, FLOAT, SMALLFLOAT
CHARACTER	CHAR, NCHAR, NVARCHAR, VARCHAR, DATE
INTERVAL	INTERVAL
DATETIME	DATETIME

Characters must be enclosed in quotation marks. Date literals must be of the format specified with the **DBDATE** environment variable. If **DBDATE** is not set, the format *mm/dd/yyyy* is assumed.

For information on using a literal INTERVAL, refer to the Literal INTERVAL segment on page 1-749. For more information on using a literal DATETIME, refer to the Literal DATETIME segment on page 1-746.

You cannot designate NULL as a default value for a column that is part of a primary key.

Data Type Requirements for Certain Columns

The following table indicates the data type requirements for columns that specify the CURRENT, DBSERVERNAME, SITENAME, TODAY, or USER functions as the default value.

Function Name	Data Type Requirement
CURRENT	DATETIME column with matching qualifier
DBSERVERNAME	CHAR, NCHAR, NVARCHAR, or VARCHAR column at least 18 characters long
SITENAME	CHAR, NCHAR, NVARCHAR, or VARCHAR column at least 18 characters long
TODAY	DATE column
USER	CHAR or VARCHAR column at least 8 characters long

Example of Default Values in Column Definitions

The following example creates a table called **accounts**. In **accounts**, the **acc_num, acc_type**, and **acc_descr** columns have literal default values. The **acc_id** column defaults to the user's login name.

```
CREATE TABLE accounts (
    acc_num INTEGER DEFAULT 0001,
    acc_type CHAR(1) DEFAULT 'A',
    acc_descr CHAR(20) DEFAULT 'New Account',
    acc_id CHAR(8) DEFAULT USER)
```

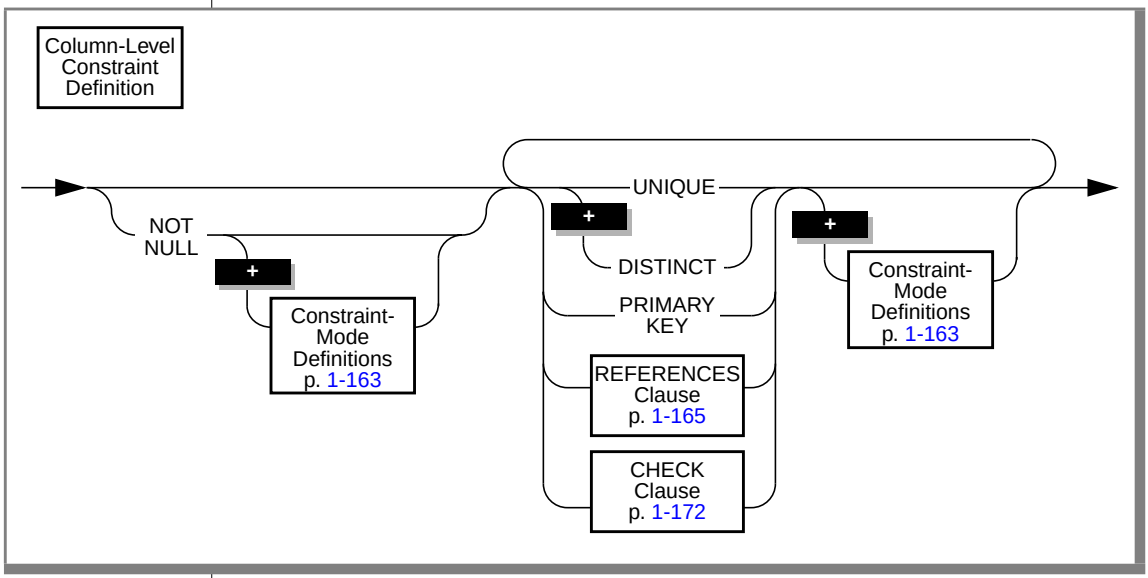
Specifying a Not Null Constraint in a Column Definition

If you do not indicate a default value for a column, the default is NULL *unless* you place a not null constraint on the column. In this case, no default value exists for the column. The following example creates the **newitems** table. In **newitems**, the column **manucode** does not have a default value nor does it allow nulls.

```
CREATE TABLE newitems (
    newitem_num INTEGER,
    manucode CHAR(3) NOT NULL,
    promotype INTEGER,
    descrip CHAR(20))
```

If you place a not null constraint on a column (and no default value is specified), you *must* enter a value into this column when you insert a row or update that column in a row. If you do not enter a value, the database server returns an error.

Column-Level Constraint-Definition Option



Unlike the table-level constraint-definition option, constraints at the column level are limited to a single column. In other words, you cannot create not null, check, unique, primary, or foreign-key multiple-column constraints. For more information on the unique, primary-key, and check constraints, see "Table-Level Constraint-Definition Option" on page 1-174.

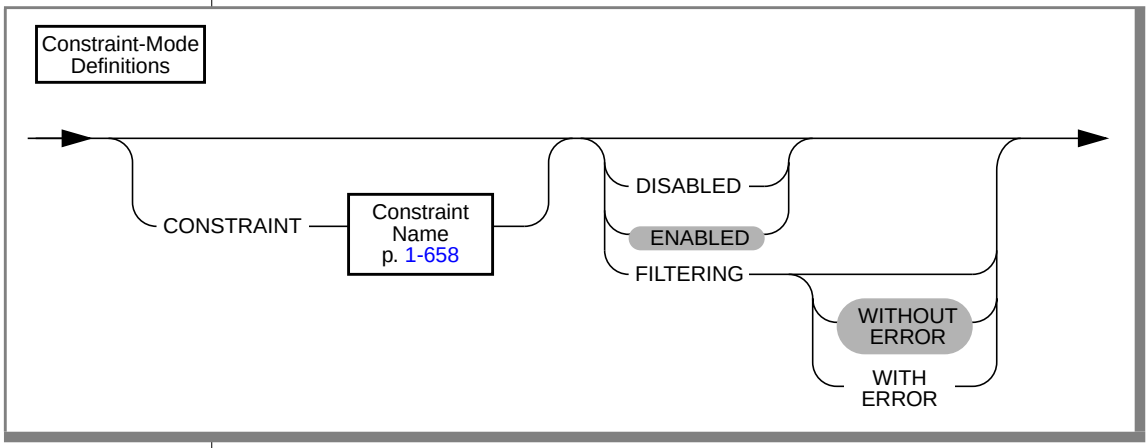
The following example creates a simple table with two constraints, a primary-key constraint named **num** on the **acc_num** column and a unique constraint named **code** on the **acc_code** column:

```
CREATE TABLE accounts (
    acc_num INTEGER PRIMARY KEY CONSTRAINT num,
    acc_code INTEGER UNIQUE CONSTRAINT code,
    acc_descr CHAR(30))
```

Using Blob Data Types in Constraints

You cannot place a unique, primary-key, or referential constraint on BYTE or TEXT columns. However, you can check for null or non-null values if you place a check constraint on a BYTE or TEXT column.

Constraint-Mode Definitions



You can use the Constraint-Mode Definitions option for the following purposes:

- You can assign a name to a column-level or table-level constraint.
- You can set any type of column-level constraint or table-level constraint to the disabled, enabled, or filtering-object modes.

Description of Constraint Modes

You can set constraints in the following modes: disabled, enabled, and filtering. The following list explains these modes and options.

Constraint Mode	Effect
disabled	A constraint created in disabled mode is not enforced during insert, delete, and update operations.
enabled	A constraint created in enabled mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement fails.
filtering	A constraint created in filtering mode is enforced during insert, delete, and update operations. If a target row causes a violation of the constraint, the statement continues processing, but the bad row is written to the violations table associated with the target table. Diagnostic information about the constraint violation is written to the diagnostics table associated with the target table.

If you choose filtering mode, you can specify the WITHOUT ERROR or WITH ERROR options. The following list explains these options.

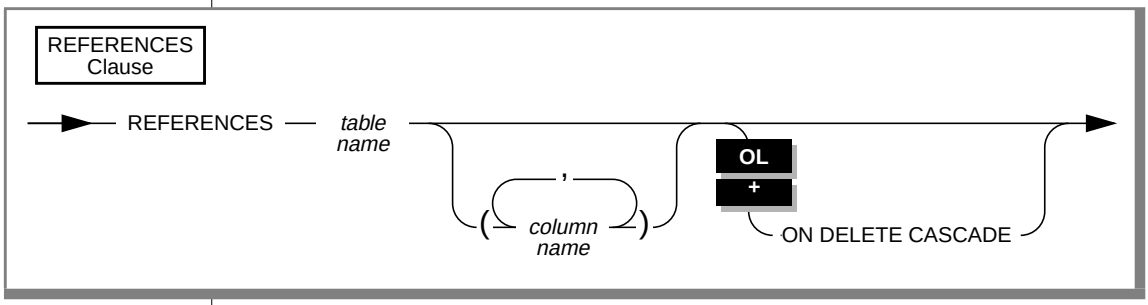
Error Option	Effect
WITHOUT ERROR	When a filtering-mode constraint is violated during an insert, delete, or update operation, no integrity-violation error is returned to the user.
WITH ERROR	When a filtering-mode constraint is violated during an insert, delete, or update operation, an integrity-violation error is returned to the user.

Using Constraint-Mode Definitions

You must observe the following rules concerning the use of constraint-mode definitions:

- If you do not specify the object mode of a column-level constraint or table-level constraint explicitly, the default mode is enabled.
- If you do not specify the WITH ERROR or WITHOUT ERROR option for a filtering-mode constraint, the default error option is WITHOUT ERROR.
- Constraints defined on temporary tables are always in the enabled mode. You cannot create a constraint on a temporary table in the disabled or filtering mode, nor can you use the SET statement to switch the object mode of a constraint on a temporary table to the disabled or filtering mode.
- You cannot assign a name to a not null constraint on a temporary table.
- You cannot create a constraint on a table that is serving as a violations or diagnostics table for another table.

The REFERENCES Clause



Element	Purpose	Restrictions	Syntax
column name	A referenced column or columns in the referenced table. If the referenced table is different from the referencing table, the default is the primary-key column or columns. If the referenced table is the same as the referencing table, there is no default.	You must observe restrictions on the table where the column resides, the existing constraints on the column, the data type of the column, and the maximum number of columns. See "Restrictions on the Column Name Variable in the REFERENCES Clause" below.	Identifier, p. 1-723
table name	The name of the referenced table	The referenced table can be the same table as the referencing table, or it can be a different table. The referenced table must reside in the same database as the referencing table.	Table Name, p. 1-768

Restrictions on the Column Name Variable in the REFERENCES Clause

You must observe the following restrictions on the *column name* variable.

Table Where the Column Resides

The referenced column (the column that you specify in the *column name* variable) can be in the same table as the referencing column, or the referenced column can be in a different table.

Existing Constraints on the Column

The referenced column must be a unique or primary-key column. That is, the referenced column in the referenced table must already have a unique or primary-key constraint placed on it.

Data Type of the Column

The data type of the referenced column must be identical to the data type of the referencing column. The only exception is that a referencing column must be INTEGER if the referenced column is SERIAL.

Number of Columns

You can specify only one column when you are using the REFERENCES clause at the column level (that is, when you are using the REFERENCES clause in the Column-Definition option).

You can specify multiple columns when you are using the REFERENCES clause at the table level (that is, when you are using the REFERENCES clause in the Constraint-Definition option).

The maximum number of columns and the total length of columns vary with the database server, as the following list describes:

- If you are using INFORMIX-OnLine Dynamic Server, the maximum number of columns is 16, and the total length of the columns cannot exceed 255 bytes.
- If you are using INFORMIX-SE, the maximum number of columns is 8, and the total length of the columns cannot exceed 120 bytes.

Using the REFERENCES Clause

You can use the REFERENCES clause to establish a referential relationship between two tables or within the same table.

The table referenced in the REFERENCES clause must reside in the same database as the created table.

Referenced and Referencing Column Requirements

In a referential relationship, the *referenced* column is a column or set of columns within a table that uniquely identifies each row in the table. In other words, the referenced column or set of columns must be a unique or primary-key constraint. If the referenced columns do not meet this criteria, the database server returns an error.

Unlike a referenced column, the *referencing* column or set of columns can contain null and duplicate values. However, every non-null value in the referencing columns must match a value in the referenced columns. When a referencing column meets this criteria, it is called a foreign key.

The relationship between referenced and referencing columns is called a *parent-child* relationship, where the parent is the referenced column (primary key) and the child is the referencing column (foreign key). This parent-child relationship is established through a referential constraint.

A referential constraint can be established between two tables or within the same table. For example, you can have an **employee** table where the **emp_no** column uniquely identifies every employee through an employee number. The **mgr_no** column in that table contains the employee number of the manager who manages that employee. In this case, **mgr_no** is the foreign key (the child) that references **emp_no,** the primary key (the parent).

A referential constraint must have a one-to-one relationship between referencing and referenced columns. In other words, if the primary key is a set of columns, then the foreign key also must be a set of columns that corresponds to the primary key. The following example creates two tables. The first table has a multiple-column primary key, and the second table has a referential constraint that references this key.

```
CREATE TABLE accounts (
    acc_num INTEGER,
    acc_type INTEGER,
    acc_descr CHAR(20),
    PRIMARY KEY (acc_num, acc_type))

CREATE TABLE sub_accounts (
    sub_acc INTEGER PRIMARY KEY,
    ref_num INTEGER NOT NULL,
    ref_type INTEGER NOT NULL,
    sub_descr CHAR(20),
    FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
        (acc_num, acc_type))
```

In this example, the foreign key of the **sub_accounts** table, **ref_num** and **ref_type**, references the primary key, **acc_num** and **acc_type**, in the **accounts** table. If, during an insert, you tried to insert a row into the **sub_accounts** table whose value for **ref_num** and **ref_type** did not exactly correspond to the values for **acc_num** and **acc_type** in an existing row in the **accounts** table, the database server would return an error. Likewise, if you attempt to update **sub_accounts** with values for **ref_num** and **ref_type** that do not correspond to an equivalent set of values in **acc_num** and **acc_type** (from the **accounts** table), the database server returns an error.

If you are referencing a primary key in another table, you do not have to state the primary-key columns in that table explicitly. Referenced tables that do not specify the referenced columns default to the primary-key columns. The references section of the previous example can be rewritten, as the following example shows:

```
.
.
.
    FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
.
.
.
```

Because **acc_num** and **acc_type** is the primary key of the **accounts** table, and no other columns are specified, the foreign key, **ref_num** and **ref_type**, references those columns.

Data Type Restrictions

The data types of the referencing and referenced columns must be identical unless the column is SERIAL data type. You can specify SERIAL for the primary key of the parent table and INTEGER for the foreign key. In the previous example, a one-to-one correspondence exists between the data types of the primary and foreign keys. If the primary-key column was defined as type SERIAL, the statement would still be successfully executed.

You cannot place a referential constraint on a BYTE or TEXT column.

Locking Implications

When you create a referential constraint, an exclusive lock is placed on the referenced table. The lock is released when the CREATE TABLE statement is done. If you are creating a table in a database with transactions, and you are using transactions, the lock is released at the end of the transaction.

Using REFERENCES in a Column Definition

When you use the REFERENCES clause at the column-definition level, you can reference a single column. The following example creates two tables, **accounts** and **sub_accounts**. A referential constraint is created between the foreign key, **ref_num**, in the **sub_accounts** table and the primary key, **acc_num**, in the **accounts** table.

```
CREATE TABLE accounts (
    acc_num INTEGER PRIMARY KEY,
    acc_type INTEGER,
    acc_descr CHAR(20))

CREATE TABLE sub_accounts (
    sub_acc INTEGER PRIMARY KEY,
    ref_num INTEGER REFERENCES accounts (acc_num),
    sub_descr CHAR(20))
```

Note that **ref_num** is not explicitly called a foreign key in the column-definition syntax. At the column level, the foreign-key designation is applied automatically.

If you are referencing the primary key in another table, you do not need to specify the referenced table column. In the preceding example, you can simply reference the **accounts** table without specifying a column. Because **acc_num** is the primary key of the **accounts** table, it becomes the referenced column by default.

Using the ON DELETE CASCADE Clause

Cascading deletes allow you to specify whether you want rows deleted in the child table when rows are deleted in the parent table. Unless you specify cascading deletes, the default prevents you from deleting data in the parent table if child tables are associated with the parent table. With the ON DELETE CASCADE clause, when you delete a row in the parent table, any rows associated with that row (foreign keys) in a child table are also deleted. The principal advantage to the cascading deletes feature is that it allows you to reduce the quantity of SQL statements you need to perform delete actions.

For example, the **all_candy** table contains the **candy_num** column as a primary key. The **hard_candy** table refers to the **candy_num** column as a foreign key. The following CREATE TABLE statement creates the **hard_candy** table with the cascading-delete clause on the foreign key:

```
CREATE TABLE hard_candy (candy_num INT, candy_flavor CHAR(20),
 FOREIGN KEY (candy_num) REFERENCES all_candy ON DELETE CASCADE);
```

With cascading deletes specified on the child table, in addition to deleting a candy item from the **all_candy** table, the delete cascades to the **hard_candy** table associated with the **candy_num** foreign key.

You specify cascading deletes with the REFERENCES clause on a column-level or table-level constraint. You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to perform cascading deletes; however, you do need the Delete privilege on tables referenced in the DELETE statement. After you indicate cascading deletes, when you delete a row from a parent table, OnLine deletes any associated matching rows from the child table.

What Happens to Multiple Children Tables

If you have a parent table with two child constraints, one child with cascading deletes specified and one child without cascading deletes, and you attempt to delete a row from the parent table that applies to both child tables, the delete statement fails, and no rows are deleted from either the parent or child tables.

Locking and Logging

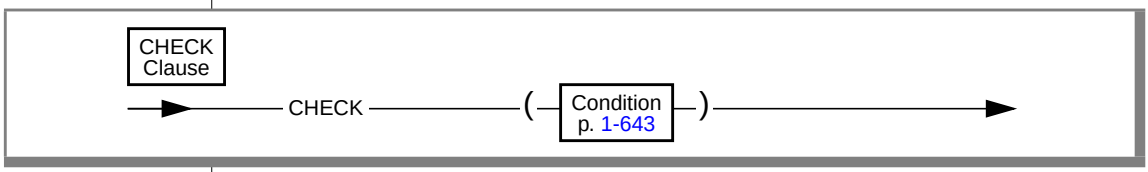
During deletes, the database server places locks on all qualifying rows of the referenced and referencing tables. You must turn logging on when you perform the deletes. If logging is turned off in a database, even temporarily, deletes do not cascade. This restriction applies because if logging is turned off, you cannot roll back any actions. For example, if a parent row is deleted, and the system crashes before the child rows are deleted, the database will have dangling child records, which violates referential integrity. However, when logging is turned back on, subsequent deletes cascade.

Restriction on Cascading Deletes

Cascading deletes can be used for most deletes. The only exception is correlated subqueries. In correlated subqueries, the subquery (or inner SELECT) is correlated when the value it produces depends on a value produced by the outer SELECT statement that contains it. If you have implemented cascading deletes, you cannot write deletes that use a child table in the correlated subquery. You receive an error when you attempt to delete from a query that contains such a correlated subquery.

See Chapter 4 of the *Informix Guide to SQL: Tutorial* for a detailed discussion about cascading deletes.

The CHECK Clause



Check constraints allow you to designate conditions that must be met *before* data can be assigned to a column during an INSERT or UPDATE statement. If a row evaluates to *false* for any check constraint defined on a table during an insert or update, the database server returns an error.

Check constraints are defined using *search conditions*. The search condition cannot contain subqueries; aggregates; host variables; rowids; the CURRENT, USER, SITENAME, DBSERVERNAME, or TODAY functions; or stored procedure calls.

Warning: *When you specify a date value in a search condition, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on how the database server interprets the search condition. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect how the database server interprets the search condition, so the check constraint might not work as you intended. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

Defining Check Constraints at the Column Level

If you define a check constraint at the column level, the only column that the check constraint can check against is the column itself. In other words, the check constraint cannot depend upon values in other columns of the table. For example, as the following statement shows, the table **acct_chk** has two columns with check constraints:

```
CREATE TABLE acct_chk (
    chk_id SERIAL PRIMARY KEY,
    debit INTEGER REFERENCES accounts (acc_num),
    debit_amt MONEY CHECK (debit_amt BETWEEN 0 AND 99999),
    credit INTEGER REFERENCES accounts (acc_num),
    credit_amt MONEY CHECK (credit_amt BETWEEN 0 AND 99999))
```

Both **debit_amt** and **credit_amt** are columns of MONEY data type whose values must be between 0 and 99999. If, however, you wanted to test that both columns contained the same value, you would not be able to create the check constraint at the column level. To create a constraint that checks values in more than one column, you must define the constraint at the table level.

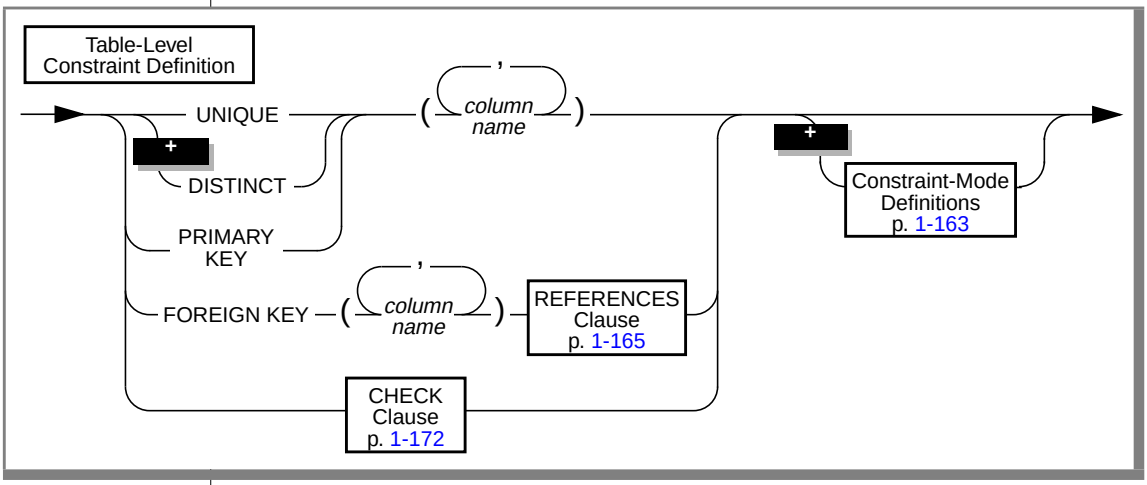
Defining Check Constraints at the Table Level

When a check constraint is defined at the table level, each column in the search condition must be a column in that table. You cannot create a check constraint for columns across tables. The next example builds the same table and columns as the previous example. However, the check constraint now spans two columns in the table.

```
CREATE TABLE acct_chk (
    chk_id SERIAL PRIMARY KEY,
    debit INTEGER REFERENCES accounts (acc_num),
    debit_amt MONEY,
    credit INTEGER REFERENCES accounts (acc_num),
    credit_amt MONEY,
    CHECK (debit_amt = credit_amt))
```

In this example, the **debit_amt** and **credit_amt** columns must equal each other, or the insert or update fails.

Table-Level Constraint-Definition Option



Element	Purpose	Restrictions	Syntax
column name	The name of the column or columns on which the constraint is placed	You must observe general restrictions that apply regardless of the type of constraint you are defining. You must also observe specific restrictions that depend on the type of constraint you are defining. See "Restrictions on the Column Name Variable in the Table-Level Constraint-Definition Option" below.	Identifier, p. 1-723

Restrictions on the Column Name Variable in the Table-Level Constraint-Definition Option

You must observe the following restrictions on the *column name* variable:

- General restrictions that apply regardless of the type of constraint you are defining
- Specific restrictions that depend on the type of constraint you are defining

General Restrictions

The column must be a column in the table, and the column cannot be a BYTE or TEXT column.

The maximum number of columns and the total length of the columns vary with the database server. If you are using INFORMIX-OnLine Dynamic Server, you can include up to 16 columns in a list of columns. The total length of all the columns cannot exceed 255 bytes. If you are using INFORMIX-SE, you can use up to 8 columns in a list of columns. The total length of all the columns cannot exceed 120 bytes.

Restrictions for Unique Constraints

When you define a unique constraint (UNIQUE or DISTINCT keywords), a column cannot appear in the constraint list more than once.

You cannot place a unique constraint on a column on which you have already placed a primary-key constraint.

Restrictions for Primary-Key Constraints

You can define a primary-key constraint (PRIMARY KEY keywords) on only one column or one set of columns in a table.You cannot define a column or set of columns as a primary key if you have already defined another column or set of columns as the primary key.

Restrictions for Referential Constraints

When you specify a referential constraint, the data type of the referencing column (the column you specify after the FOREIGN KEY keywords) must match the data type of the referenced column (the column you specify in the REFERENCES clause). The only exception is that the referencing column must be INTEGER if the referenced column is SERIAL.

Using the Table-Level Constraint-Definition Option

The table-level constraint-definition option allows you to create constraints for a single column or a set of columns. You can create the following types of constraints with this option: unique, primary-key, foreign-key (referential), and check constraints.

Defining a Column as Unique

Use the UNIQUE keyword to require that a single column or set of columns accepts only unique data. You cannot insert duplicate values in a column that has a unique constraint.

Each column named in a unique constraint must be a column in the table and cannot appear in the constraint list more than once. The following example creates a simple table that has a unique constraint on one of its columns:

```
CREATE TABLE accounts (a_name CHAR(12), a_code SERIAL,
    UNIQUE (a_name) CONSTRAINT  acc_name)
```

If you want to define the constraint at the column level instead, simply include the keywords UNIQUE and CONSTRAINT in the column definition, as the following example shows:

```
CREATE TABLE accounts
   (a_name CHAR(12) UNIQUE CONSTRAINT acc_name, a_code SERIAL)
```

You cannot place a unique constraint on a BYTE or TEXT column.

Defining a Column as a Primary Key

A primary key is a column or set of columns that contains a non-null unique value for each row in a table. A table can have only one primary key, and a column that is defined as a primary key cannot also be defined as unique. In the previous two examples, a unique constraint was placed on the column **a_name**. The following example creates this column as the primary key for the **accounts** table:

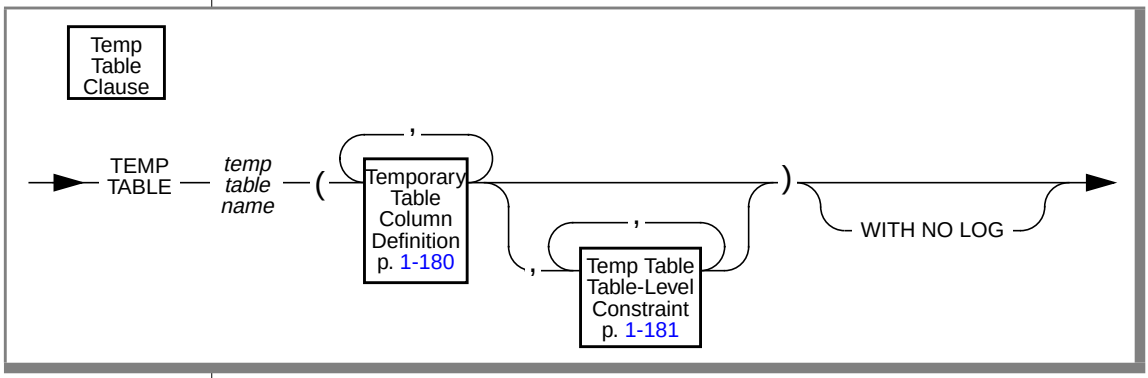
```
CREATE TABLE accounts
    (a_name CHAR(12), a_code SERIAL, PRIMARY KEY (a_name))
```

You cannot place a primary-key constraint on a BYTE or TEXT column.

Defining a Column as a Foreign Key

A foreign key *joins* and establishes dependencies between tables. A foreign key references a unique or primary key in a table. For every entry in the foreign-key columns, a matching entry must exist in the unique or primary-key columns if all foreign-key columns contain non-null values. You cannot make BYTE or TEXT columns foreign keys.

TEMP TABLE Clause



Element	Purpose	Restrictions	Syntax
temp table name	The name that you want to assign to the temporary table	The name must be different from any existing table, view, or synonym name in the current database, but it does not have to be different from other temporary table names used by other users.	Identifier, p. 1-723

Temporary tables created with the CREATE TEMP TABLE statement are called *explicit* temporary tables. Explicit temporary tables can also be created with the SELECT ... INTO TEMP statement.

When an application creates an explicit temporary table, it exists until one of the following situations occurs:

- The application terminates.
- The application closes the database where the table was created. In this case, the table is dropped only if the database does transaction logging, and the temporary table was not created with the WITH NO LOG option.
- The application closes the database where the table was created and opens a database in a different database server (a second OnLine or an SE database server).

When any of these events occur, the temporary table is deleted.

DB

The INFO statement and the Info Menu Option cannot be used with temporary tables. ♦

Temporary table names must be different from existing table, view, or synonym names in the current database; however, they need not be different from other temporary table names used by other users.

Temporary tables that are created as a part of processing are called *implicit* temporary tables. Implicit temporary tables are discussed in the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

You can specify where temporary tables are created with the CREATE TEMP TABLE statement, environment variables, and ONCONFIG parameters. OnLine stores temporary tables in the following order:

1. The IN *dbspace* clause

   You can specify the dbspace where you want the temporary table stored with the IN *dbspace* clause of the CREATE TABLE statement.

2. The dbspaces you specify when you fragment temporary tables

   Use the FRAGMENT BY clause of the CREATE TABLE statement to fragment regular and temporary tables.

3. The **DBSPACETEMP** environment variable

   If you do not use the IN *dbspace* clause or the FRAGMENT BY clause to fragment the table, OnLine checks to see if the **DBSPACETEMP** environment variable is set. The **DBSPACETEMP** environment variable lists dbspaces where temporary tables can be stored. This list can include standard dbspaces, temporary dbspaces, or both. If the environment variable is set, OnLine stores the temporary table in one of the dbspaces specified in that list.

4. The ONCONFIG parameter DBSPACETEMP

   You can specify a location for temporary tables with the ONCONFIG parameter DBSPACETEMP.

If you do not use the IN *dbspace* clause, the FRAGMENT BY clause to fragment the table, the **DBSPACETEMP** environment variable, or the ONCONFIG parameter DBSPACETEMP, the temporary tables are created in the same dbspace as your database server.

You can specify more than one dbspace for the **DBSPACETEMP** environment variable. For example, you can specify the following dbspace definitions for the **DBSPACETEMP** environment variable:

```
setenv DBSPACETEMP tempspc1:tempspc2:tempspc3
```

Each temporary table that you create round-robins to a dbspace. For example, if you created three temporary tables, the first one, **temp1**, would go into the dbspace called **tempspc1**; the second one, **temp2**, would go into **tempspc2**; and the third one, **temp3**, would go into **tempspc3**.

SE

Temporary tables are created in the directory specified by the **DBTEMP** environment variable. If the **DBTEMP** environment variable is not set, temporary tables are created in the directory of the database (that is, the **.dbs** directory). ♦

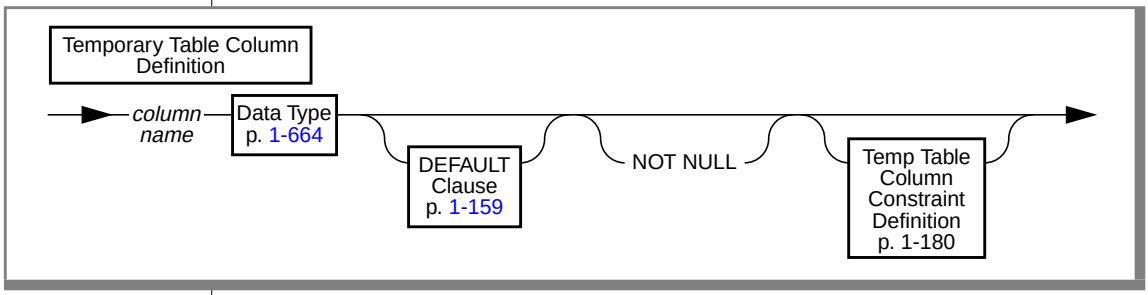
For additional information about the **DBSPACETEMP** environment variable, see Chapter 4, "Environment Variables" in the *Informix Guide to SQL: Reference*. For additional information about the ONCONFIG parameter DBSPACETEMP, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

If you have the Connect privilege on a database, you can create temporary tables. Once a temporary table is created, you can build indexes on the table. However, you are the only user who can see the temporary table.

Fragmenting Temporary Tables

You can create temporary tables with a fragmentation strategy.

Temporary-Table Column Definition

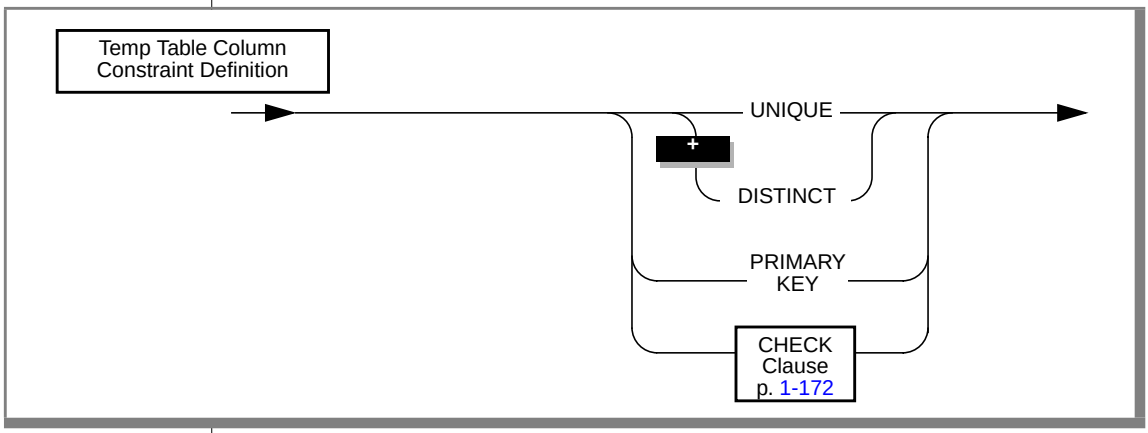


Element	Purpose	Restrictions	Syntax
column name	The name of a column in the table	Name must be unique within a table, but you can use the same names in different tables in the same database.	Identifier, p. 1-723

You define columns for temporary tables in the same manner as you define columns for regular database tables. The only difference is the option for defining column constraints, which is defined in the following section.

For more information about defining single columns for temporary tables, see the "Column-Definition Option" on page 1-158.

Temporary-Table Column Constraint Definition

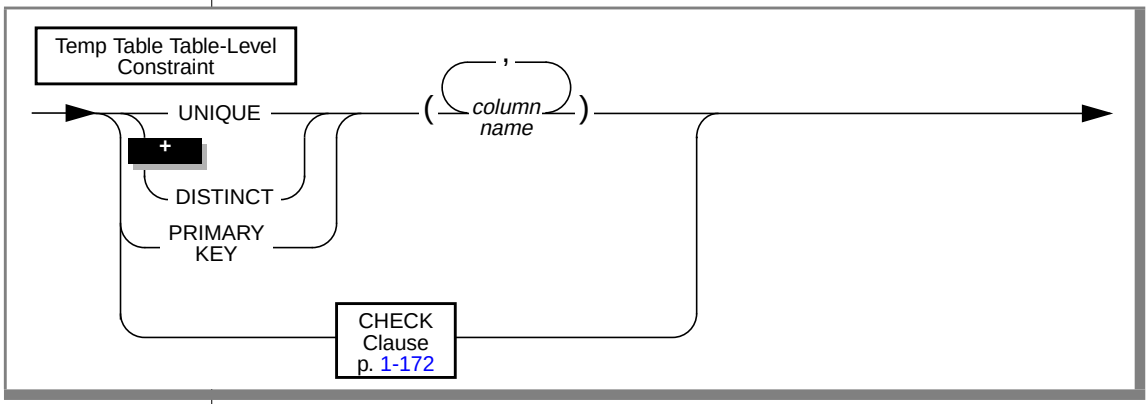


Temporary-table column constraints are the same as column constraints for regular tables, with the following exceptions:

- You cannot place referential constraints on columns in a temporary table. Temporary columns cannot be referenced or referencing columns.
- The Constraint Mode Definitions option is not available for columns in temporary tables. You cannot assign a name to a constraint on a temporary-table column. You cannot set the object mode of a constraint on a temporary-table column. See "Constraint-Mode Definitions" on page 1-163 for further information on this option.

For more information about column constraints in regular tables, see "Column-Level Constraint-Definition Option" on page 1-162.

Table-Level Constraint for Temporary Tables



Element	Purpose	Restrictions	Syntax
column name	The name of the column or columns on which the constraint is placed	See "Restrictions on Table-Level Constraints for Temporary Tables" on page 1-182.	Identifier, p. 1-723

You can place a table-level constraint on one or more columns of a temporary table.

Restrictions on Table-Level Constraints for Temporary Tables

Table-level constraints are defined for temporary tables in the same manner as regular database tables, with the following exceptions:

- You cannot place referential constraints on columns in a temporary table. In other words, temporary columns cannot be referenced or referencing columns.
- The Constraint Mode Definitions option is not available for constraints on columns in temporary tables. You cannot assign a name to a constraint on a temporary-table column. You cannot set the object mode of a constraint on a temporary-table column. See "Constraint-Mode Definitions" on page 1-163 for further information on this option.

For more information about table-level constraints on regular tables, see "Table-Level Constraint-Definition Option" on page 1-174.

WITH NO LOG Option for Temporary Tables

You must use the WITH NO LOG keywords on temporary tables created in temporary dbspaces.

Using the WITH NO LOG keywords prevents logging of temporary tables in databases started with logging.

If you use the WITH NO LOG keywords in a CREATE TABLE statement, and the database does not use logging, the WITH NO LOG option is ignored.

Once you turn off logging on a temporary table, you cannot turn it back on; a temporary table is, therefore, always logged or never logged.

The following example shows how to prevent logging temporary tables in a database that uses logging:

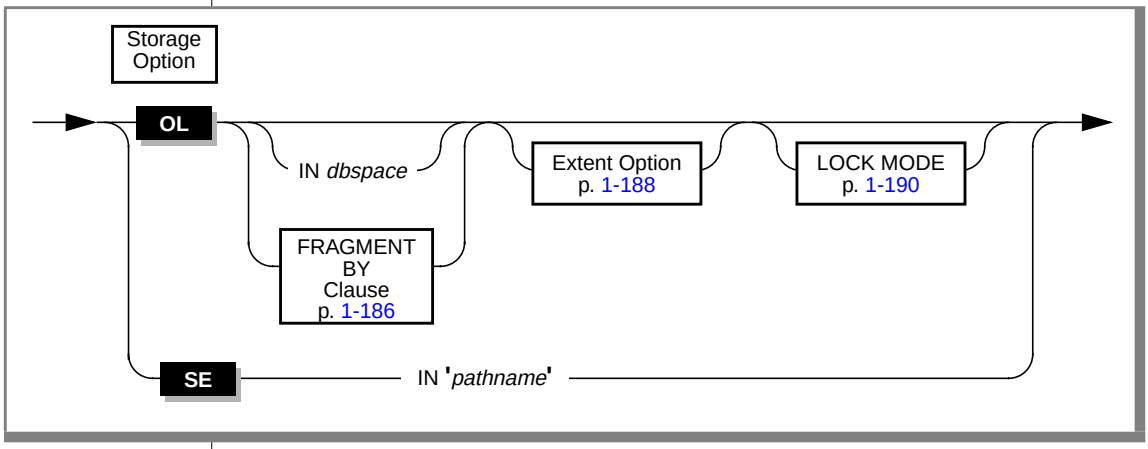
```
CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15))
    WITH NO LOG
```

The WITH ROWIDS Clause

Important: *Use the WITH ROWIDS clause only on fragmented tables. In non-fragmented tables, the rowid column remains unchanged. Informix recommends, however, that you utilize primary keys as an access method rather than exploiting the rowid column.*

Nonfragmented tables contain a hidden column called the rowid column. However, fragmented tables do not contain this column. If a table is fragmented, you can use the WITH ROWIDS clause to add the rowid column to the table. OnLine assigns each row in the rowid column a unique number that remains stable for the life of the row. The database server uses an index to find the physical location of the row. Each row contains an additional 4 bytes to store the rowid column after you add it.

Storage Option



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace in which a database table or temporary table is to be stored. The default for database tables is the dbspace in which the current database resides.	Specified dbspace must already exist.	Identifier, p. 1-723
pathname	The full operating-system pathname and filename in which you want to store the database table. The default is the directory of the database (the **.dbs** directory).	You cannot use the *pathname* variable for a temporary table. You can specify any valid directory in *pathname*. You cannot add an extension to the filename.	See "The IN pathname Option" on page 1-190.

The storage option allows you to specify where the table is stored and the locking granularity for the table.

The IN dbspace Clause

The IN *dbspace* clause allows you to isolate a table. The dbspace that you specify must already exist. If you do not specify the IN *dbspace* clause, the default is the dbspace where the current database resides. Temporary tables do not have a default dbspace. For further information about storing temporary tables, see the "TEMP TABLE Clause" on page 1-177.

For example, if the **stores7** database is in the **stockdata** dbspace, but you want the **customer** data placed in a separate dbspace called **custdata**, use the following statements:

```
CREATE DATABASE stores7 IN stockdata

CREATE TABLE customer
    (
    customer_num    SERIAL(101),
    fname           CHAR(15),
    lname           CHAR(15),
    company         CHAR(20),
    address1        CHAR(20),
    address2        CHAR(20),
    city            CHAR(15),
    state           CHAR(2),
    zipcode         CHAR(5),
    phone           CHAR(18)
    )
 IN custdata EXTENT SIZE 16

    .
    .
    .
```

For more information about storing your tables in separate dbspaces, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

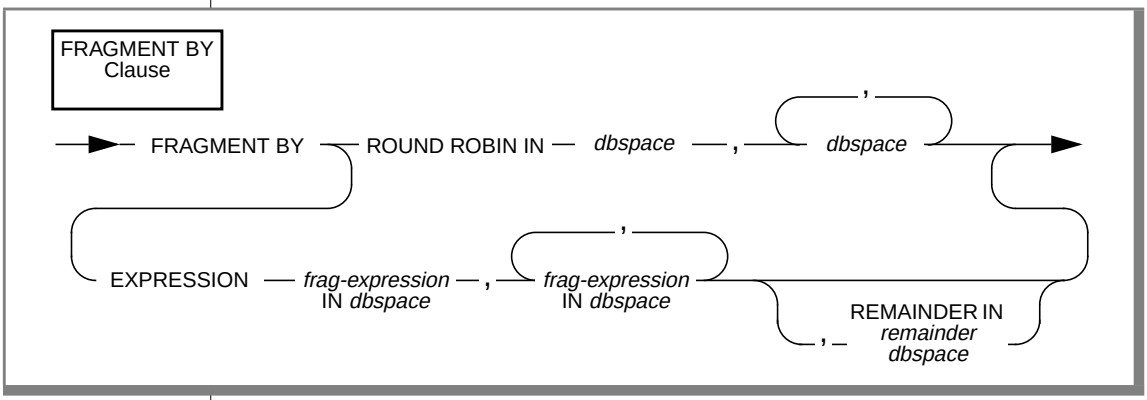
If your table has one or more blob columns, you can store the blob data with the table data or in a separate blobspace. See the Data Type segment on page 1-664 for more information. The following example shows how blobspaces and dbspaces are specified.

The following statement creates the **resume** table. The data for the table is stored in the **employ** dbspace. The data in the **resume** column is stored with the table, but the data associated with the **photo** column is stored in a blobspace named **photo_space**.

```
CREATE TABLE resume
    (
    fname           CHAR(15),
    lname           CHAR(15),
    phone           CHAR(18),
    recd_date       DATETIME YEAR TO HOUR,
    contact_date    DATETIME YEAR TO HOUR,
    comments        VARCHAR(250, 100),
    vita            TEXT IN TABLE,
    photo           BYTE IN photo_space
    )
IN employ
```

The FRAGMENT BY Clause

The FRAGMENT BY clause allows you to create fragmented tables. Fragmentation means that groups of rows within a table are stored together in the same dbspace.



Element	Purpose	Restrictions	Syntax
dbspace	The dbspace that contains a table fragment	You must specify at least two dbspaces. You can specify a maximum of 2,048 dbspaces. The dbspaces must exist when you execute the statement.	Identifier, p. 1-723
frag-expression	An expression that defines a fragment where a row is to be stored using a range, hash, or arbitrary rule	If you specify a value for *remainder dbspace*, you must specify at least one fragment expression. If you do not specify a value for *remainder dbspace*, you must specify at least two fragment expressions. You can specify a maximum of 2,048 fragment expressions. Each fragment expression can contain only columns from the current table and only data values from a single row. No subqueries, stored procedures, current date / time functions, or aggregates are allowed in a fragment expression.	Expression, p. 1-671, and Condition, p. 1-643

(1 of 2)

Element	Purpose	Restrictions	Syntax
remainder dbspace	The dbspace that contains table rows that do not meet the conditions defined in any fragment expression	If you specify two or more fragment expressions, *remainder dbspace* is optional. If you specify only one fragment expression, *remainder dbspace* is required. The dbspace specified in *remainder dbspace* must exist at the time you execute the statement.	Identifier, p. 1-723

(2 of 2)

Use the FRAGMENT BY clause to define the distribution scheme, either round-robin or expression-based.

In a round-robin distribution scheme, specify at least two dbspaces where you want the fragments to be placed. As records are inserted into the table, they are placed in the first available dbspace. OnLine balances the load between the specified dbspaces as you insert records and distributes the rows in such a way that the fragments always maintain approximately the same number of rows. In this distribution scheme, the database server must scan all fragments when searching for a row.

In an *expression-based* distribution scheme, each fragment expression in a rule specifies a dbspace. Each fragment expression within the rule isolates data and aids the database server in searching for rows. Specify one of the following rules:

- Range rule

  A range rule specifies fragment expressions that use a range to specify which rows are placed in a fragment, as the following example shows:

```
...
FRAGMENT BY EXPRESSION
c1 < 100 IN dbsp1,
c1 >= 100 and c1 < 200 IN dbsp2,
c1 >= 200 IN dbsp3;
```

- Hash rule

  A hash rule specifies fragment expressions that are created when you use a hash algorithm, which is often implemented with the MOD function, as the following example shows:

```
...
FRAGMENT BY EXPRESSION
MOD(id_num, 3) = 0 IN dbsp1,
MOD(id_num, 3) = 1 IN dbsp2,
MOD(id_num, 3) = 2 IN dbsp3;
```

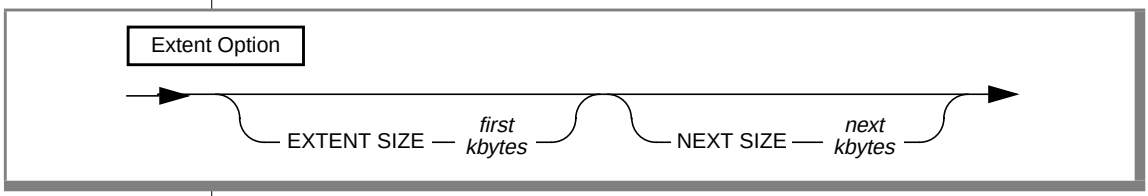
- Arbitrary rule

  An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically includes the use of OR clauses to group data, as the following example shows:

```
...
FRAGMENT BY EXPRESSION
zip_num = 95228 OR zip_num = 95443 IN dbsp2,
zip_num = 91120 OR zip_num = 92310 IN dbsp4,
REMAINDER IN dbsp5;
```

Warning: *When you specify a date value in a fragment expression, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on the distribution scheme. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect the distribution scheme and can produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

Extent Option



Element	Purpose	Restrictions	Syntax
first kbytes	The length in kilobytes of the first extent for the table. The default length is eight times the disk page size on your system. For example, if you have a 2-kilobyte page system, the default length is 16 kilobytes.	The minimum length is four times the disk page size on your system. For example, if you have a 2-kilobyte page system, the minimum length is eight kilobytes. The maximum length is equal to the chunk size.	Expression, p. 1-671
next kbytes	The length in kilobytes for the subsequent extents. The default length is eight times the disk page size on your system. For example, if you have a 2-kilobyte page system, the default length is 16 kilobytes.	The minimum length is four times the disk page size on your system. For example, if you have a 2-kilobyte page system, the minimum length is 8 kilobytes.The maximum length is equal to the chunk size.	Expression, p. 1-671

See the *INFORMIX-OnLine Dynamic Server Performance Guide* for a discussion about calculating extent sizes.

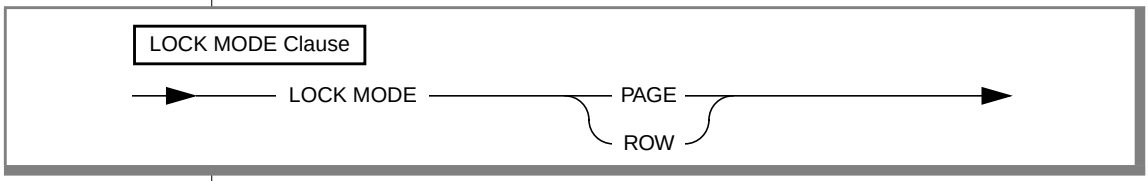
The following example specifies a first extent of 20 kilobytes and allows the rest of the extents to use the default size:

```
CREATE TABLE emp_info
    (
    f_name CHAR(20),
    l_name CHAR(20),
    position CHAR(20),
    start_date DATETIME YEAR TO DAY,
    comments VARCHAR(255)
    )
EXTENT SIZE 20
```

Revising Extent Sizes for Unloaded Tables

You can revise the CREATE TABLE statements in generated schema files to revise the extent and next-extent sizes of unloaded tables. See the *INFORMIX-OnLine Dynamic Server Administrator's Guide* for information about revising extent sizes.

LOCK MODE Clause



The default locking granularity is a page.

Row-level locking provides the highest level of concurrency. However, if you are using many rows at one time, the lock-management overhead can become significant. You can also exceed the maximum number of locks available, depending on the configuration of your OnLine system.

Page locking allows you to obtain and release one lock on a whole page of rows. Page locking is especially useful when you know that the rows are grouped into pages in the same order that you are using to process all the rows. For example, if you are processing the contents of a table in the same order as its cluster index, page locking is especially appropriate.

You can change the lock mode of an existing table with the ALTER TABLE statement.

The IN pathname Option

SE

The *pathname* in an IN clause can specify any valid directory and is not restricted to the directory that contains the current database. This allows you to spread your tables over multiple disks.

In UNIX, *pathname* cannot be longer than 64 characters and must be within quotes (' '). A pathname must appear in the following form:

```
[/directory-name/...]filename
```

If *pathname* in an IN clause specifies a filename that is different from *table name*, always use *table name* (rather than the filename) to refer to the table in subsequent <vk>SQL statements.

The creator of the table must have search permissions on all directories in the path and write permissions on the directory that is to contain the files. ♦

References

See the ALTER TABLE, CREATE INDEX, CREATE DATABASE, DROP TABLE, and SET statements in this manual. Also see the Condition, Data Type, Identifier, and Table Name segments.

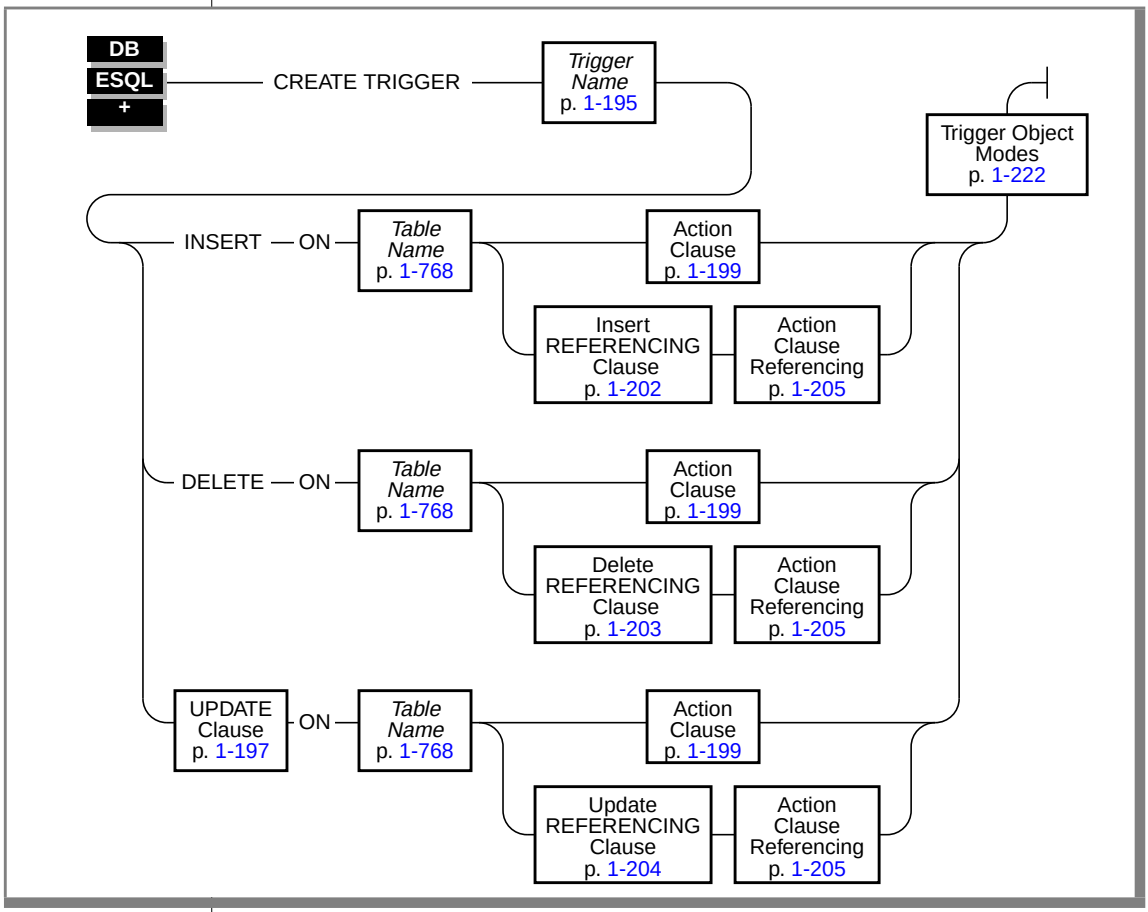
In the *Informix Guide to SQL: Tutorial*, see the discussion of data-integrity constraints and the discussion of the ON DELETE CASCADE clause in Chapter 4. Also see the discussion of creating a database and tables in Chapter 9.

In the *INFORMIX-OnLine Dynamic Server Performance Guide*, see the discussion of extent sizing.

CREATE TRIGGER

Use the CREATE TRIGGER statement to create a trigger on a table in the database. A trigger is a database object that automatically sets off a specified set of SQL statements when a specified event occurs.

Syntax



Usage

You must be either the owner of the table or have the DBA status to create a trigger on a table.

You can use roles with triggers. Role-related statements (CREATE ROLE, DROP ROLE, and SET ROLE) and SET SESSION AUTHORIZATION statements can be triggered inside a trigger. Privileges that a user has acquired through enabling a role or through a SET SESSION AUTHORIZATION statement are not relinquished when a trigger is executed.

You can define a trigger with a stand-alone CREATE TRIGGER statement.

DB

You can define a trigger as part of a schema by placing the CREATE TRIGGER statement inside a CREATE SCHEMA statement. ♦

You can create a trigger only on a table in the current database. You cannot create a trigger on a temporary table, a view, or a system catalog table.

You cannot create a trigger inside a stored procedure if the procedure is called inside a data manipulation statement. For example, you cannot create a trigger inside the stored procedure **sp_items** in the following INSERT statement:

```
INSERT INTO items EXECUTE PROCEDURE sp_items
```

See "Data Manipulation Statements" on page 1-13 for a list of data manipulation statements.

ESQL

If you are embedding the CREATE TRIGGER statement in an ESQL/C or ESQL/COBOL program, you cannot use a host variable in the trigger specification. ♦

You cannot use a stored procedure variable in a CREATE TRIGGER statement.

SE

You cannot use a ROLLBACK WORK statement to undo a CREATE TRIGGER statement. If you roll back a transaction that contains a CREATE TRIGGER statement, the trigger remains, and you do not receive an error message. ♦

The Trigger Event

The trigger event specifies the type of statement that activates a trigger. The trigger event can be an INSERT, DELETE, or UPDATE statement. Each trigger can have only one trigger event. The occurrence of the trigger event is the *triggering statement*.

For each table, you can define only one trigger that is activated by an INSERT statement and only one trigger that is activated by a DELETE statement. For each table, you can define multiple triggers that are activated by UPDATE statements. See "UPDATE Clause" on page 1-197 for more information about multiple triggers on the same table.

You cannot define a DELETE trigger event on a table with a referential constraint that specifies ON DELETE CASCADE.

You are responsible for guaranteeing that the triggering statement returns the same result with and without the triggered actions. See "Action Clause" on page 1-199 and "Triggered Action List" on page 1-206 for more information on the behavior of triggered actions.

If INFORMIX-OnLine Dynamic Server is the database server, a triggering statement from an external database server can activate the trigger. As shown in the following example, an insert trigger on **newtab**, managed by **dbserver1**, is activated by an INSERT statement from **dbserver2**. The trigger executes as if the insert originated on **dbserver1**.

```
-- Trigger on stores7@dbserver1:newtab

CREATE TRIGGER ins_tr INSERT ON newtab
REFERENCING new AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE nt_pct (post_ins.mc));

-- Triggering statement from dbserver2

INSERT INTO stores7@dbserver1:newtab
    SELECT item_num, order_num, quantity, stock_num,
manu_code,
    total_price FROM items;
```

Trigger Events with Cursors

If the triggering statement uses a cursor, the complete trigger is activated each time the statement executes. For example, if you declare a cursor for a triggering INSERT statement, each PUT statement executes the complete trigger. Similarly, if a triggering UPDATE or DELETE statement contains the clause WHERE CURRENT OF, each update or delete activates the complete trigger. This behavior is different from what occurs when a triggering statement does not use a cursor and updates multiple rows. In this case, the set of triggered actions executes only once. For more information on the execution of triggered actions, see "Action Clause" on page 1-199.

Privileges on the Trigger Event

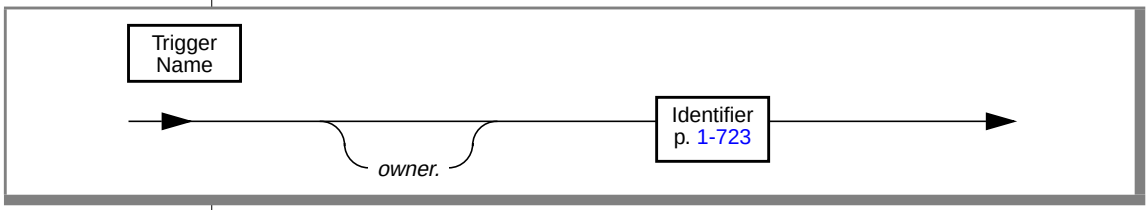
You must have the appropriate Insert, Delete, or Update privilege on the triggering table to execute the INSERT, DELETE, or UPDATE statement that is the trigger event. The triggering statement might still fail, however, if you do not have the privileges necessary to execute one of the SQL statements in the action clause. When the triggered actions are executed, the database server checks your privileges for each SQL statement in the trigger definition as if the statement were being executed independently of the trigger. For information on the privileges you need to execute a trigger, see "Privileges to Execute Triggered Actions" on page 1-215.

Impact of Triggers

The INSERT, DELETE, and UPDATE statements that initiate triggers might appear to execute slowly because they activate additional SQL statements, and the user might not know that other actions are occurring.

The execution time for a triggering data manipulation statement depends on the complexity of the triggered action and whether it initiates other triggers. Obviously, the elapsed time for the triggering data manipulation statement increases as the number of cascading triggers increases. For more information on triggers that initiate other triggers, see "Cascading Triggers" on page 1-216.

Trigger Name



Element	Purpose	Restrictions	Syntax
owner	The user name of the owner of the trigger	The specified name must be a valid user name.	Identifier, p. 1-723

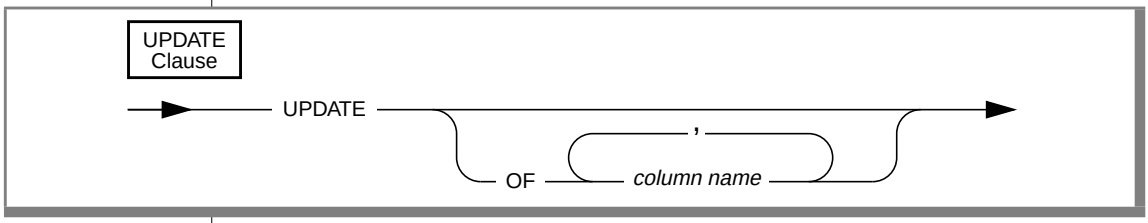
When you create a trigger, the name of the trigger must be unique within a database.

ANSI

When you create a trigger, the *owner.name* combination (the combination of the owner name and trigger name) must be unique within a database. ♦

For information about the relationship between the trigger owner's privileges and the privileges of other users, see "Privileges to Execute Triggered Actions" on page 1-215.

UPDATE Clause



Element	Purpose	Restrictions	Syntax
column name	The name of a column or columns that activate the trigger. The default is all the columns in the table on which you create the trigger.	The specified columns must belong to the table on which you create the trigger. If you define more than one update trigger on a table, the column lists of the triggering statements must be mutually exclusive.	Identifier, p. 1-723

If the trigger event is an UPDATE statement, the trigger executes when any column in the triggering column list is updated.

If the triggering UPDATE statement updates more than one of the triggering columns in a trigger, the trigger executes only once.

Defining Multiple Update Triggers

If you define more than one update trigger event on a table, the column lists of the triggers must be mutually exclusive. The following example shows that **trig3** is illegal on the **items** table because its column list includes **stock_num**, which is a triggering column in **trig1**. Multiple update triggers on a table cannot include the same columns.

```
CREATE TRIGGER trig1 UPDATE OF item_num, stock_num ON items
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW(EXECUTE PROCEDURE proc1());

CREATE TRIGGER trig2 UPDATE OF manu_code ON items
BEFORE(EXECUTE PROCEDURE proc2());

-- Illegal trigger: stock_num occurs in trig1
CREATE TRIGGER trig3 UPDATE OF order_num, stock_num ON items
BEFORE(EXECUTE PROCEDURE proc3());
```

When an UPDATE Statement Activates Multiple Triggers

When an UPDATE statement updates multiple columns that have different triggers, the column numbers of the triggering columns determine the order of trigger execution. Execution begins with the smallest triggering column number and proceeds in order to the largest triggering column number. The following example shows that table **taba** has four columns (**a**, **b**, **c**, **d**):

```
CREATE TABLE taba (a int, b int, c int, d int)
```

Define **trig1** as an update on columns **a** and **c**, and define **trig2** as an update on columns **b** and **d**, as shown in the following example:

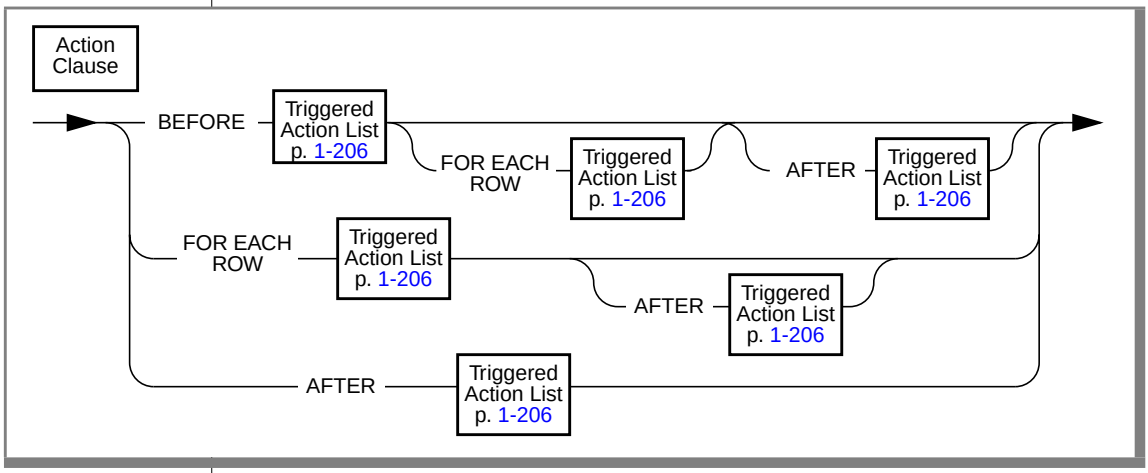
```
CREATE TRIGGER trig1 UPDATE OF a, c ON taba
    AFTER (UPDATE tabb SET y = y + 1);

CREATE TRIGGER trig2 UPDATE OF b, d ON taba
    AFTER (UPDATE tabb SET z = z + 1);
```

The triggering statement is shown in the following example:

```
UPDATE taba SET (b, c) = (b + 1, c + 1)
```

Then **trig1** for columns **a** and **c** executes first, and **trig2** for columns **b** and **d** executes next. In this case, the smallest column number in the two triggers is column 1 (**a**), and the next is column 2 (**b)**.

Action Clause



The action clause defines the characteristics of triggered actions and specifies the time when these actions occur. You must define at least one triggered action, using the keywords BEFORE, FOR EACH ROW, or AFTER to indicate when the action occurs relative to the triggering statement. You can specify triggered actions for all three options on a single trigger, but you must order them in the following sequence: BEFORE, FOR EACH ROW, and AFTER. You cannot follow a FOR EACH ROW triggered action list with a BEFORE triggered action list. If the first triggered action list is FOR EACH ROW, an AFTER action list is the only option that can follow it. See "Action Clause Referencing" on page 1-205 for more information on the action clause when a REFERENCING clause is present.

BEFORE Actions

The BEFORE triggered action or actions execute once before the triggering statement executes. If the triggering statement does not process any rows, the BEFORE triggered actions still execute because the database server does not yet know whether any row is affected.

FOR EACH ROW Actions

The FOR EACH ROW triggered action or actions execute once for each row that the triggering statement affects. The triggered SQL statement executes after the triggering statement processes each row.

If the triggering statement does not insert, delete, or update any rows, the FOR EACH ROW triggered actions do not execute.

AFTER Actions

An AFTER triggered action or actions execute once after the action of the triggering statement is complete. If the triggering statement does not process any rows, the AFTER triggered actions still execute.

Actions of Multiple Triggers

When an UPDATE statement activates multiple triggers, the triggered actions merge. Assume that **taba** has columns **a**, **b**, **c**, and **d**, as shown in the following example:

```
CREATE TABLE taba (a int, b int, c int, d int)
```

Next, assume that you define **trig1** on columns **a** and **c**, and **trig2** on columns **b** and **d**. If both triggers have triggered actions that are executed BEFORE, FOR EACH ROW, and AFTER, then the triggered actions are executed in the following sequence:

1. BEFORE action list for trigger (**a**, **c**)
2. BEFORE action list for trigger (**b**, **d**)
3. FOR EACH ROW action list for trigger (**a**, **c**)
4. FOR EACH ROW action list for trigger (**b**, **d**)
5. AFTER action list for trigger (**a**, **c**)
6. AFTER action list for trigger (**b**, **d**)

The database server treats the triggers as a single trigger, and the triggered action is the merged-action list. All the rules governing a triggered action apply to the merged list as one list, and no distinction is made between the two original triggers.

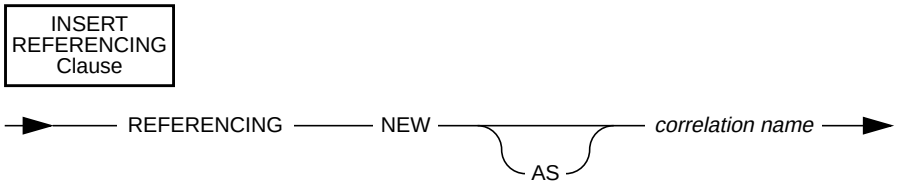
Guaranteeing Row-Order Independence

In a FOR EACH ROW triggered-action list, the result might depend on the order of the rows being processed. You can ensure that the result is independent of row order by following these suggestions:

- Avoid selecting the triggering table in the FOR EACH ROW section. If the triggering statement affects multiple rows in the triggering table, the result of the SELECT statement in the FOR EACH ROW section varies as each row is processed. This condition also applies to any cascading triggers. See "Cascading Triggers" on page 1-216.
- In the FOR EACH ROW section, avoid updating a table with values derived from the current row of the triggering table. If the triggered actions modify any row in the table more than once, the final result for that row depends on the order in which rows from the triggering table are processed.
- Avoid modifying a table in the FOR EACH ROW section that is selected by another triggered statement in the same FOR EACH ROW section, including any cascading triggered actions. If you modify a table in this section and refer to it later, the changes to the table might not be complete when you refer to it. Consequently, the result might differ, depending on the order in which rows are processed.

The database server does not enforce rules to prevent these situations because doing so would restrict the set of tables from which a triggered action can select. Furthermore, the result of most triggered actions is independent of row order. Consequently, you are responsible for ensuring that the results of the triggered actions are independent of row order.

INSERT REFERENCING Clause



Element	Purpose	Restrictions	Syntax
correlation name	A name that you assign to a new column value so that you can refer to it within the triggered action. The new column value in the triggering table is the value of the column after execution of the triggering statement.	The correlation name must be unique within the CREATE TRIGGER statement.	Identifier, p. 1-723

Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See "Action Clause Referencing" on page 1-205.

To use the correlation name, precede the column name with the correlation name, followed by a period. For example, if the new correlation name is **post**, refer to the new value for the column **fname** as **post.fname**.

If the trigger event is an INSERT statement, using the old correlation name as a qualifier causes an error because no value exists before the row is inserted. For the rules that govern the use of correlation names, see "Using Correlation Names in Triggered Actions" on page 1-209.

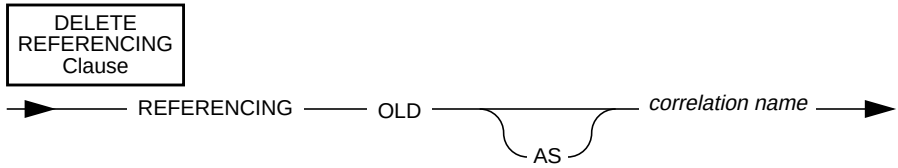
You can use the INSERT REFERENCING clause only if you define a FOR EACH ROW triggered action.

The following example illustrates the use of the INSERT REFERENCING clause. This example inserts a row into **backup_table1** for every row that is inserted into **table1**. The values that are inserted into **col1** and **col2** of **backup_table1** are an exact copy of the values that were just inserted into **table1**.

```
CREATE TABLE table1 (col1 INT, col2 INT);
CREATE TABLE backup_table1 (col1 INT, col2 INT);
CREATE TRIGGER before_trig
    INSERT ON table1
    REFERENCING NEW as new
    FOR EACH ROW
    (
    INSERT INTO backup_table1 (col1, col2)
    VALUES (new.col1, new.col2)
    );
```

As the preceding example shows, the advantage of the INSERT REFERENCING clause is that it allows you to refer to the data values that the trigger event in your triggered action produces.

DELETE REFERENCING Clause



Element	Purpose	Restrictions	Syntax
correlation name	A name that you assign to an old column value so that you can refer to it within the triggered action. The old column value in the triggering table is the value of the column before execution of the triggering statement.	The correlation name must be unique within the CREATE TRIGGER statement.	Identifier, p. 1-723

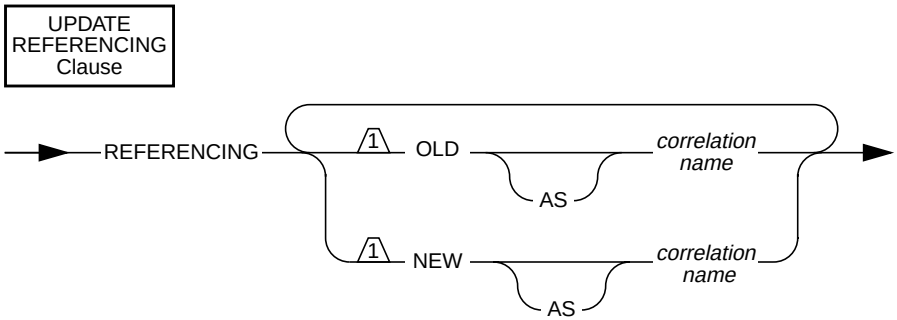
Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See "Action Clause Referencing" on page 1-205.

You use the correlation name to refer to an old column value by preceding the column name with the correlation name and a period (.). For example, if the old correlation name is **pre**, refer to the old value for the column **fname** as **pre.fname**.

If the trigger event is a DELETE statement, using the new correlation name as a qualifier causes an error because the column has no value after the row is deleted. See "Using Correlation Names in Triggered Actions" on page 1-209 for the rules governing the use of correlation names.

You can use the DELETE REFERENCING clause only if you define a FOR EACH ROW triggered action.

UPDATE REFERENCING Clause



Element	Purpose	Restrictions	Syntax
correlation name	A name that you assign to an old or new column value so that you can refer to it within the triggered action. The old column value in the triggering table is the value of the column before execution of the triggering statement. The new column value in the triggering table is the value of the column after executing the triggering statement.	You can specify a correlation name for an old column value only (OLD option), for a new column value only (NEW option), or for both the old and new column values. Each correlation name you specify must be unique within the CREATE TRIGGER statement.	Identifier, p. 1-723

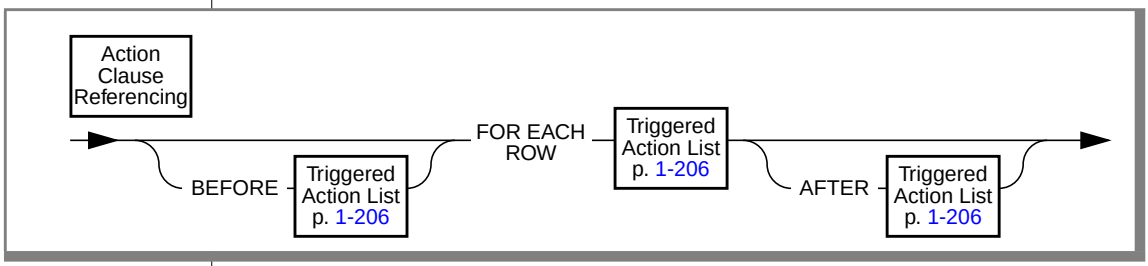
Once you assign a correlation name, you can use it only inside the FOR EACH ROW triggered action. See "Action Clause Referencing" on page 1-205.

Use the correlation name to refer to an old or new column value by preceding the column name with the correlation name and a period (.). For example, if the new correlation name is **post**, you refer to the new value for the column **fname** as **post.fname**.

If the trigger event is an UPDATE statement, you can define both old and new correlation names to refer to column values before and after the triggering update. See "Using Correlation Names in Triggered Actions" on page 1-209 for the rules that govern the use of correlation names.

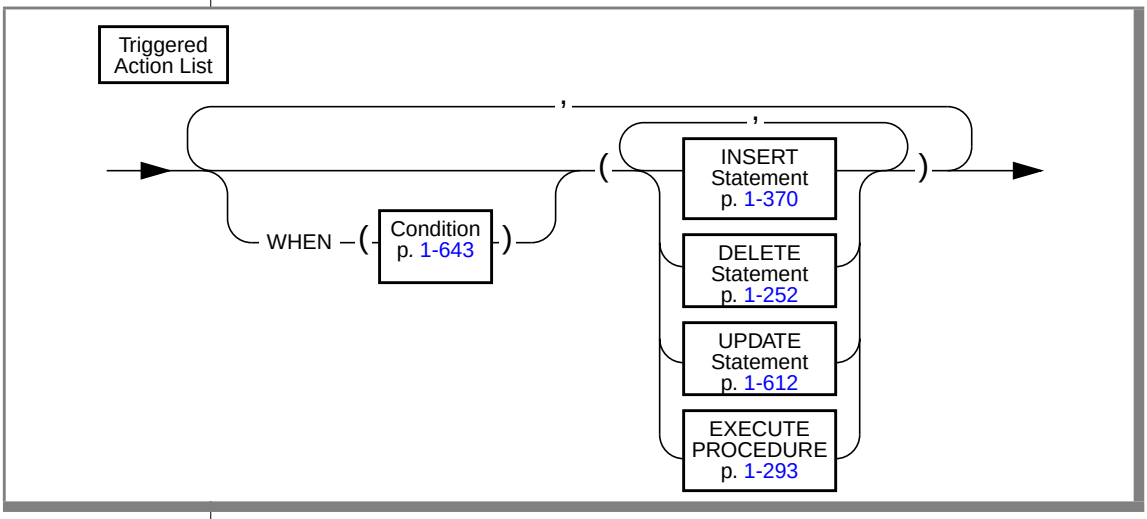
You can use the UPDATE REFERENCING clause only if you define a FOR EACH ROW triggered action.

Action Clause Referencing



If the CREATE TRIGGER statement contains an INSERT REFERENCING clause, a DELETE REFERENCING clause, or an UPDATE REFERENCING clause, you *must* include a FOR EACH ROW triggered-action list in the action clause. You can also include BEFORE and AFTER triggered-action lists, but they are optional. See "Action Clause" on page 1-199 for information on the BEFORE, FOR EACH ROW, and AFTER triggered-action lists.

Triggered Action List



The triggered action consists of an optional WHEN condition and the action statements. Objects that are referenced in the triggered action, that is, tables, columns, and stored procedures, must exist when the CREATE TRIGGER statement is executed. This rule applies only to objects that are referenced directly in the trigger definition.

Warning: *When you specify a date expression in the WHEN condition or in an action statement, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 4-digit year, the* ***DBCENTURY*** *environment variable has no effect on how the database server interprets the date expression. When you specify a 2-digit year, the* ***DBCENTURY*** *environment variable can affect how the database server interprets the date expression, so the triggered action might produce unpredictable results. See the "Informix Guide to SQL: Reference" for more information on the* ***DBCENTURY*** *environment variable.*

The WHEN Condition

The WHEN condition lets you make the triggered action dependent on the outcome of a test. When you include a WHEN condition in a triggered action, if the triggered action evaluates to *true*, the actions in the triggered action list execute in the order in which they appear. If the WHEN condition evaluates to *false* or *unknown*, the actions in the triggered action list are not executed. If the triggered action is in a FOR EACH ROW section, its search condition is evaluated for each row.

For example, the triggered action in the following trigger executes only if the condition in the WHEN clause is true:

```
CREATE TRIGGER up_price
UPDATE OF unit_price ON stock
REFERENCING OLD AS pre NEW AS post
FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
    (INSERT INTO warn_tab VALUES(pre.stock_num,
        pre.order_num, pre.unit_price, post.unit_price,
        CURRENT))
```

A stored procedure that executes inside the WHEN condition carries the same restrictions as a stored procedure that is called in a data manipulation statement. See "CREATE PROCEDURE" on page 1-134 for more information about a stored procedure that is called within a data manipulation statement.

The Action Statements

The triggered-action statements can be INSERT, DELETE, UPDATE, or EXECUTE PROCEDURE statements. If a triggered-action list contains multiple statements, these statements execute in the order in which they appear in the list.

SE

In INFORMIX-SE, all objects referenced in the triggered actions must be in the current database. ♦

Achieving a Consistent Result

To guarantee that the triggering statement returns the same result with and without the triggered actions, make sure that the triggered actions in the BEFORE and FOR EACH ROW sections do not modify any table referenced in the following clauses:

- WHERE clause
- SET clause in the UPDATE statement
- SELECT clause
- EXECUTE PROCEDURE clause in a multiple-row INSERT statement

Using Keywords

If you use the INSERT, DELETE, UPDATE, or EXECUTE keywords as an identifier in any of the following clauses inside a triggered action list, you must qualify them by the owner name, the table name, or both:

- FROM clause of a SELECT statement
- INTO clause of the EXECUTE PROCEDURE statement
- GROUP BY clause
- SET clause of the UPDATE statement

You get a syntax error if these keywords are *not* qualified when you use these clauses inside a triggered action.

If you use the keyword as a column name, it must be qualified by the table name—for example, **table.update**. If both the table name and the column name are keywords, they must be qualified by the owner name—for example, **owner.insert.update**. If the owner name, table name, and column name are all keywords, the owner name must be in quotes—for example, **'delete'.insert.update**. The only exception is when these keywords are the first table or column name in the list, and you do not have to qualify them. For example, **delete** in the following statement does not need to be qualified because it is the first column listed in the INTO clause:

```
CREATE TRIGGER t1 UPDATE OF b ON tab1
    FOR EACH ROW (EXECUTE PROCEDURE p2()
    INTO delete, d)
```

The following statements show examples in which you must qualify the column name or the table name:

FROM clause of a SELECT statement

```
CREATE TRIGGER t1 INSERT ON tab1
    BEFORE (INSERT INTO tab2 SELECT * FROM tab3,
    'owner1'.update)
```

INTO clause of an EXECUTE PROCEDURE statement

```
CREATE TRIGGER t3 UPDATE OF b ON tab1
    FOR EACH ROW (EXECUTE PROCEDURE p2() INTO
    d, tab1.delete)
```

GROUP BY clause of a SELECT statement

```
CREATE TRIGGER t4 DELETE ON tab1
    BEFORE (INSERT INTO tab3 SELECT deptno, SUM(exp)
    FROM budget GROUP BY deptno, budget.update)
```

SET clause of an UPDATE statement

```
CREATE TRIGGER t2 UPDATE OF a ON tab1
    BEFORE (UPDATE tab2 SET a = 10, tab2.insert = 5)
```

Using Correlation Names in Triggered Actions

The following rules apply when you use correlation names in triggered actions:

- You can use the correlation names for the old and new column values only in statements in the FOR EACH ROW triggered-action list. You can use the old and new correlation names to qualify any column in the triggering table in either the WHEN condition or the triggered SQL statements.
- The old and new correlation names refer to all rows affected by the triggering statement.

- You cannot use the correlation name to qualify a column name in the GROUP BY, the SET, or the COUNT DISTINCT clause.
- The scope of the correlation names is the entire trigger definition. This scope is statically determined, meaning that it is limited to the trigger definition; it does not encompass cascading triggers or columns that are qualified by a table name in a stored procedure that is a triggered action.

When to Use Correlation Names

In an SQL statement in a FOR EACH ROW triggered action, you must qualify all references to columns in the triggering table with either the old or new correlation name, unless the statement is valid independent of the triggered action.

In other words, if a column name inside a FOR EACH ROW triggered action list is not qualified by a correlation name, even if it is qualified by the triggering table name, it is interpreted as if the statement is independent of the triggered action. No special effort is made to search the definition of the triggering table for the nonqualified column name.

For example, assume that the following DELETE statement is a triggered action inside the FOR EACH ROW section of a trigger:

```
DELETE FROM tab1 WHERE col_c = col_c2
```

For the statement to be valid, both **col_c** and **col_c2** must be columns from **tab1**. If **col_c2** is intended to be a correlation reference to a column in the triggering table, it must be qualified by either the old or the new correlation name. If **col_c2** is not a column in **tab1** and is not qualified by either the old or new correlation name, you get an error.

When a column is not qualified by a correlation name, and the statement is valid independent of the triggered action, the column name refers to the current value in the database. In the triggered action for trigger **t1** in the following example, **mgr** in the WHERE clause of the correlated subquery is an unqualified column from the triggering table. In this case, **mgr** refers to the current column value in **empsal** because the INSERT statement is valid independent of the triggered action.

```
CREATE DATABASE db1;
CREATE TABLE empsal (empno INT, salary INT, mgr INT);
CREATE TABLE mgr (eno INT, bonus INT);
CREATE TABLE biggap (empno INT, salary INT, mgr INT);

CREATE TRIGGER t1 UPDATE OF salary ON empsal
AFTER (INSERT INTO biggap SELECT * FROM empsal WHERE salary <
    (SELECT bonus FROM mgr WHERE eno = mgr));
```

In a triggered action, an unqualified column name from the triggering table refers to the current column value, but only when the triggered statement is valid independent of the triggered action.

Qualified Versus Unqualified Value

The following table summarizes the value retrieved when you use the column name qualified by the old correlation name and the column name qualified by the new correlation name.

Trigger Event	old.col	new.col
INSERT	no value (error)	inserted value
UPDATE (column updated)	original value	current value (N)
UPDATE (column not updated)	original value	current value (U)
DELETE	original value	no value (error)

Refer to the following key when you read the table.

Term	Meaning
original value	is the value before the triggering statement.
current value	is the value after the triggering statement.
(N)	cannot be changed by triggered action.
(U)	can be updated by triggered statements; value may be different from original value because of preceding triggered actions.

Outside a FOR EACH ROW triggered-action list, you cannot qualify a column from the triggering table with either the old correlation name or the new correlation name; it always refers to the current value in the database.

Action on the Triggering Table

You cannot reference the triggering table in any triggered SQL statement, with the following exceptions:

- The trigger event is UPDATE and the triggered SQL statement is also UPDATE, and the columns in both statements, including any nontriggering columns in the triggering UPDATE, are mutually exclusive.

  For example, assume that the following UPDATE statement, which updates columns **a** and **b** of **tab1**, is the triggering statement:

  ```
  UPDATE tab1 SET (a, b) = (a + 1, b + 1)
  ```

  Now consider the triggered actions in the following example. The first UPDATE statement is a valid triggered action, but the second one is not because it updates column **b** again.

  ```
  UPDATE tab1 SET c = c + 1; -- OK
  UPDATE tab1 SET b = b + 1;-- ILLEGAL
  ```

- The triggered SQL statement is a SELECT statement. The SELECT statement can be a triggered statement in the following instances:
  - The SELECT statement appears in a subquery in the WHEN clause or a triggered-action statement.
  - The triggered action is a stored procedure, and the SELECT statement appears inside the stored procedure.

This rule, which states that a triggered SQL statement cannot reference the triggering table, with the two noted exceptions, applies recursively to all cascading triggers, which are considered part of the initial trigger. This situation means that a cascading trigger cannot update any columns in the triggering table that were updated by the original triggering statement, including any nontriggering columns affected by that statement. For example, assume the following UPDATE statement is the triggering statement:

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1)
```

Then in the cascading triggers shown in the following example, **trig2** fails at runtime because it references column **b**, which is updated by the triggering UPDATE statement. See "Cascading Triggers" on page 1-216 for more information about cascading triggers.

```
CREATE TRIGGER trig1 UPDATE OF a ON tab1-- Valid
    AFTER (UPDATE tab2 set e = e + 1);

CREATE TRIGGER trig2 UPDATE of e ON tab2-- Invalid
    AFTER (UPDATE tab1 set b = b + 1);
```

Rules for Stored Procedures

The following rules apply to a stored procedure that is used as a triggered action:

- The stored procedure cannot be a cursory procedure (that is, a procedure that returns more than one row) in a place where only one row is expected.
- When an EXECUTE PROCEDURE statement is the triggered action, you can specify the INTO clause only for an UPDATE trigger when the triggered action occurs in the FOR EACH ROW section. In this case, the INTO clause can contain only column names from the triggering table. The following statement illustrates the appropriate use of the INTO clause:

```
CREATE TRIGGER upd_totpr UPDATE OF quantity ON items
REFERENCING OLD AS pre_upd NEW AS post_upd
FOR EACH ROW(EXECUTE PROCEDURE
 calc_totpr(pre_upd.quantity,
 post_upd.quantity, pre_upd.total_price)
 INTO total_price)
```

  When the INTO clause appears in the EXECUTE PROCEDURE statement, the database server updates the columns named there with the values returned from the stored procedure. The database server performs the update immediately upon returning from the stored procedure. See "EXECUTE PROCEDURE" on page 1-293 for more information about the statement.
- You cannot use the old or new correlation name inside the stored procedure. If you need to use the corresponding values in the procedure, you must pass them as parameters. The stored procedure should be independent of triggers, and the old or new correlation name do not have any meaning outside the trigger.
- You cannot use the following statements inside the stored procedure: ALTER FRAGMENT, ALTER INDEX, ALTER OPTICAL, ALTER TABLE, BEGIN WORK, COMMIT WORK, CREATE TRIGGER, DELETE, DROP INDEX, DROP OPTICAL, DROP SYNONYM, DROP TABLE, DROP TRIGGER, DROP VIEW, INSERT, RENAME COLUMN, RENAME TABLE, ROLLBACK WORK, SET CONSTRAINTS, and UPDATE.

When you use a stored procedure as a triggered action, the objects that it references are not checked until the procedure is executed.

Privileges to Execute Triggered Actions

If you are not the trigger owner but the trigger owner's privileges include the WITH GRANT OPTION privilege, you inherit the owner's privileges as well as the WITH GRANT OPTION privilege for each triggered SQL statement. You have these privileges in addition to your privileges.

If the triggered action is a stored procedure, you must have the Execute privilege on the procedure or the owner of the trigger must have the Execute privilege and the WITH GRANT OPTION privilege. Inside the stored procedure, you do not carry the privileges of the trigger owner; instead you have the following privileges:

1. The triggered action is a DBA-privileged procedure.

   When you are granted the Execute privilege on the procedure, the database server automatically grants you DBA privileges for the procedure execution. These DBA privileges are available only when you are executing the procedure.

2. The triggered action is an owner-privileged procedure.

   If the procedure owner has the WITH GRANT OPTION right for the necessary privileges on the underlying objects, you inherit these privilege when you are granted the Execute privilege. In this case, all the nonqualified objects that the procedure references are qualified by the name of the procedure owner.

   If the procedure owner does not have the WITH GRANT OPTION right, you have your original privileges on the underlying objects when the procedure executes.

For more information on privileges on stored procedures, see Chapter 12 in the *Informix Guide to SQL: Tutorial*.

Creating a Triggered Action That Anyone Can Use

To create a trigger that is executable by anyone who has the privileges to execute the triggering statement, you can ask the DBA to create a DBA-privileged procedure and grant you the Execute privilege with the WITH GRANT OPTION right. You then use the DBA-privileged procedure as the triggered action. Anyone can execute the triggered action because the DBA-privileged procedure carries the WITH GRANT OPTION right. When you activate the procedure, the database server applies privilege-checking rules for a DBA. For more information about privileges on stored procedures, see Chapter 12 of the *Informix Guide to SQL: Tutorial*.

Cascading Triggers

The database server allows triggers to cascade, meaning that the triggered actions of one trigger can activate another trigger. The maximum number of triggers in a cascading sequence is 61; the initial trigger plus a maximum of 60 cascading triggers. When the number of cascading triggers in a series exceeds the maximum, the database server returns error number -748, as the following example shows:

```
Exceeded limit on maximum number of cascaded triggers.
```

The following example illustrates a series of cascading triggers that enforce referential integrity on the **manufact**, **stock**, and **items** tables in the **stores7** database. When a manufacturer is deleted from the **manufact** table, the first trigger, **del_manu**, deletes all the items from that manufacturer from the **stock** table. Each delete in the **stock** table activates a second trigger, **del_items**, that deletes all the **items** from that manufacturer from the **items** table. Finally, each delete in the **items** table triggers the stored procedure **log_order**, which creates a record of any orders in the **orders** table that can no longer be filled.

```
CREATE TRIGGER del_manu
DELETE ON manufact
REFERENCING OLD AS pre_del
FOR EACH ROW(DELETE FROM stock
    WHERE manu_code = pre_del.manu_code);

CREATE TRIGGER del_stock
DELETE ON stock
REFERENCING OLD AS pre_del
FOR EACH ROW(DELETE FROM items
```

```
        WHERE manu_code = pre_del.manu_code);

    CREATE TRIGGER del_items
    DELETE ON items
    REFERENCING OLD AS pre_del
    FOR EACH ROW(EXECUTE PROCEDURE log_order(pre_del.order_num));
```

When you are not using logging, or you are using the INFORMIX-SE database server, with or without logging, referential integrity constraints on both the **manufact** and **stock** tables would prohibit the triggers in this example from executing. When you use INFORMIX-OnLine Dynamic Server with logging, however, the triggers execute successfully because constraint checking is deferred until all the triggered actions are complete, including the actions of cascading triggers. See "Constraint Checking" on page 1-217 for more information about how constraints are handled when triggers execute.

The database server prevents loops of cascading triggers by not allowing you to modify the triggering table in any cascading triggered action, except an UPDATE statement, which does not modify any column that the triggering UPDATE statement updated.

Constraint Checking

When you use logging, INFORMIX-OnLine Dynamic Server defers constraint checking on the triggering statement until after the statements in the triggered-action list execute. OnLine effectively executes a SET statement (SET CONSTRAINTS ALL DEFERRED) before it executes the triggering statement. After the triggered action is completed, it effectively executes another SET statement (SET CONSTRAINTS *constr_name* IMMEDIATE) to check the constraints that were deferred. This action allows you to write triggers so that the triggered action can resolve any constraint violations that the triggering statement creates. For more information, see the SET statement on page 1-501.

Consider the following example, in which the table **child** has constraint **r1**, which references the table **parent**. You define trigger **trig1** and activate it with an INSERT statement. In the triggered action, **trig1** checks to see if **parent** has a row with the value of the current **cola** in **child**; if not, it inserts it.

```
CREATE TABLE parent (cola INT PRIMARY KEY);
CREATE TABLE child (cola INT REFERENCES parent CONSTRAINT r1);
CREATE TRIGGER trig1 INSERT ON child
    REFERENCING NEW AS new
    FOR EACH ROW
    WHEN((SELECT COUNT (*) FROM parent
        WHERE cola = new.cola) = 0)
-- parent row does not exist
    (INSERT INTO parent VALUES (new.cola));
```

When you insert a row into a table that is the child table in a referential constraint, the row might not exist in the parent table. The database server does not immediately return this error on a triggering statement. Instead, it allows the triggered action to resolve the constraint violation by inserting the corresponding row into the parent table. As the previous example shows, you can check within the triggered action to see whether the parent row exists, and if so, bypass the insert.

For an INFORMIX-OnLine Dynamic Server database without logging, OnLine does *not* defer constraint checking on the triggering statement. In this case, it immediately returns an error if the triggering statement violates a constraint.

OnLine does not allow the SET statement in a triggered action. OnLine checks this restriction when you activate a trigger because the statement could occur inside a stored procedure.

SE

For an INFORMIX-SE database, with or without logging, constraint checking occurs prior to the triggered action. If a constraint violation results from the triggering statement, INFORMIX-SE returns an error immediately. ♦

Preventing Triggers from Overriding Each Other

When you activate multiple triggers with an UPDATE statement, a trigger can possibly override the changes that an earlier trigger made. If you do not want the triggered actions to interact, you can split the UPDATE statement into multiple UPDATE statements, each of which updates an individual column. As another alternative, you can create a single update trigger for all columns that require a triggered action. Then, inside the triggered action, you can test for the column being updated and apply the actions in the desired order. This approach, however, is different than having the database server apply the actions of individual triggers, and it has the following disadvantages:

- If the trigger has a BEFORE action, it applies to all columns because you cannot yet detect whether a column has changed.
- If the triggering UPDATE statement sets a column to the current value, you cannot detect the update, so the triggered action is skipped. You might want to execute the triggered action even though the value of the column has not changed.

The Client/Server Environment

In an OnLine database, the statements inside the triggered action can affect tables in external databases. The following example shows an update trigger on **dbserver1**, which triggers an update to **items** on **dbserver2**:

```
CREATE TRIGGER upd_nt UPDATE ON newtab
REFERENCING new AS post
FOR EACH ROW(UPDATE stores7@dbserver2:items
    SET quantity = post.qty WHERE stock_num = post.stock
    AND manu_code = post.mc)
```

If a statement from an external database server initiates the trigger, however, and the triggered action affects tables in an external database, the triggered actions fail. For example, the following combination of triggered action and triggering statement results in an error when the triggering statement executes:

```
-- Triggered action from dbserver1 to dbserver3:

CREATE TRIGGER upd_nt UPDATE ON newtab
REFERENCING new AS post
FOR EACH ROW(UPDATE stores7@dbserver3:items
    SET quantity = post.qty WHERE stock_num = post.stock
    AND manu_code = post.mc);

-- Triggering statement from dbserver2:

UPDATE stores7@dbserver1:newtab
    SET qty = qty * 2 WHERE s_num = 5
    AND mc = 'ANZ';
```

SE

In an INFORMIX-SE database, all objects referenced in the triggered actions must be in the current database. ♦

Logging and Recovery

You can create triggers for databases, with and without logging. However, when the database does not have logging, you cannot roll back when the triggering statement fails. In this case, you are responsible for maintaining data integrity in the database.

In INFORMIX-OnLine Dynamic Server, if the trigger fails and the database has transactions, all triggered actions and the triggering statement are rolled back because the triggered actions are an extension of the triggering statement. The rest of the transaction, however, is not rolled back.

SE

In INFORMIX-SE, if you explicitly begin a transaction, you must explicitly roll back the whole transaction. If the database has no transactions, data integrity might possibly be violated when the triggered actions fail.

Even if the database has logging, any data definition statement in the triggered action cannot be rolled back. Again, you are responsible for maintaining data integrity as well as integrity of the database structure. ♦

The row action of the triggering statement occurs before the triggered actions in the FOR EACH ROW section. If the triggered action fails for a database without logging, the application must restore the row that was changed by the triggering statement to its previous value.

When you use a stored procedure as a triggered action, if you terminate the procedure in an exception-handling section, any actions that modify data inside that section are rolled back along with the triggering statement. In the following partial example, when the exception handler traps an error, it inserts a row into the table **logtab**:

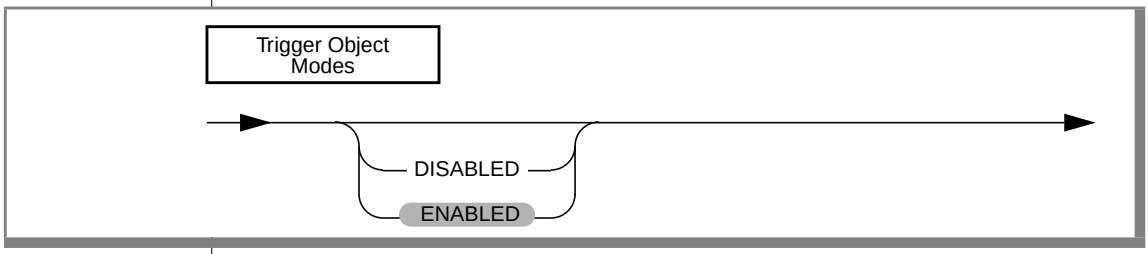
```
ON EXCEPTION IN (-201)
    INSERT INTO logtab values (errno, errstr);
    RAISE EXCEPTION -201
END EXCEPTION
```

When the RAISE EXCEPTION statement returns the error, however, the database server rolls back this insert because it is part of the triggered actions. If the procedure is executed outside a triggered action, the insert is not rolled back.

The stored procedure that implements a triggered action cannot contain any BEGIN WORK, COMMIT WORK, or ROLLBACK WORK statements. If the database has logging, you must either begin an explicit transaction before the triggering statement, or the statement itself must be an implicit transaction. In any case, another transaction-related statement cannot appear inside the stored procedure.

You can use triggers to enforce referential actions that the database server does not currently support. For any INFORMIX-SE database or for an INFORMIX-OnLine Dynamic Server database without logging, you are responsible for maintaining data integrity when the triggering statement fails.

Trigger Object Modes



The Trigger Object Modes option allows you to create a trigger in either the enabled or disabled object mode.

You can create triggers in the following object modes.

Object Mode	Effect
disabled	When a trigger is created in disabled mode, the database server does not execute the triggered action when the trigger event (an insert, delete, or update operation) takes place. In effect, the database server ignores the trigger even though its catalog information is maintained.
enabled	When a trigger is created in enabled mode, the database server executes the triggered action when the trigger event (an insert, delete, or update operation) takes place.

Specifying Object Modes for Triggers

You must observe the following rules when you specify the object mode for a trigger in the CREATE TRIGGER statement:

- If you do not specify the disabled or enabled object modes explicitly, the default object mode is enabled.
- In contrast to unique indexes and constraints of all types, you cannot set triggers to the filtering object mode because a trigger does not impose any type of data-integrity requirement on the tables in the database.

- You can use the SET statement to switch the mode of a disabled trigger to the enabled mode. Once the trigger has been re-enabled, the database server executes the triggered action whenever the trigger event takes place. However, the re-enabled trigger does not perform retroactively. The database server does not attempt to execute the trigger for rows that were inserted, deleted, or updated after the trigger was disabled and before it was enabled; therefore, be cautious about disabling a trigger. If disabling a trigger will eventually destroy the semantic integrity of the database, do not disable the trigger in the first place.
- You cannot create a trigger on a violations table or a diagnostics table.

References

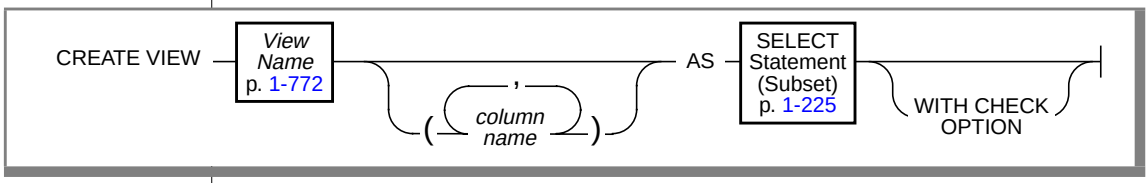
See the DROP TRIGGER, CREATE PROCEDURE, and EXECUTE PROCEDURE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see Chapter 12.

CREATE VIEW

Use the CREATE VIEW statement to create a new view that is based upon existing tables and views in the database.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column or columns in the view	See "Naming View Columns" on page 1-226.	Identifier, p. 1-723

Usage

Except for the statements in the following list, you can use a view in any <vk>SQL statement where you can use a table.

ALTER FRAGMENT	DROP TABLE
ALTER INDEX	DROP TRIGGER
ALTER TABLE	LOCK TABLE
CREATE INDEX	RECOVER TABLE
CREATE TABLE	RENAME TABLE
CREATE TRIGGER	UNLOCK TABLE
DROP INDEX	

The view behaves like a table that is called *view name*. It consists of the set of rows and columns that the SELECT statement returns each time the SELECT statement is executed by using the view. The view reflects changes to the underlying tables with one exception. If a SELECT * clause defines the view, the view has only the columns in the underlying tables at the time the view is created. New columns that are subsequently added to the underlying tables with the ALTER TABLE statement do not appear in the view.

The view name must be unique; that is, a view name cannot have the same name as another database object, such as a table, synonym, or temporary table.

The view inherits the data types of the columns from the tables from which they come. Data types of virtual columns are determined from the nature of the expression.

To create a view, you must have the Select privilege on all columns from which the view is derived.

The SELECT statement is stored in the **sysviews** system catalog table. When you subsequently refer to a view in another statement, the database server performs the defining SELECT statement while it executes the new statement.

SE

You cannot use a ROLLBACK WORK statement to undo a CREATE VIEW statement. If you roll back a transaction that contains a CREATE VIEW statement, the view remains, and you do not receive an error message. ♦

DB

If you create a view outside the CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**. ♦

Subset of a SELECT Allowed in CREATE VIEW

The SELECT statement has the form that is described on page 1-459, but in CREATE VIEW, it cannot have an ORDER BY clause, INTO TEMP clause, or UNION operator. Do not use display labels in the select list; display labels are interpreted as column names.

Naming View Columns

The number of columns that you specify in the *column name* parameter must match the number of columns returned by the SELECT statement that defines the view.

If you do not specify a list of columns, the view inherits the column names of the underlying tables. In the following example, the view **herostock** has the same column names as the ones in the SELECT statement:

```
CREATE VIEW herostock AS
    SELECT stock_num, description, unit_price, unit, unit_descr
        FROM stock WHERE manu_code = 'HRO'
```

If the SELECT statement returns an expression, the corresponding column in the view is called a *virtual* column. You must provide a name for virtual columns. You must also provide a column name in cases where the selected columns have duplicate column names when the table prefixes are stripped. For example, when both **orders.order_num** and **items.order_num** appear in the SELECT statement, you must provide two separate column names to label them in the CREATE VIEW statement, as the following example shows:

```
CREATE VIEW someorders (custnum,ocustnum,newprice) AS
    SELECT orders.order_num,items.order_num,
            items.total_price*1.5
        FROM orders, items
        WHERE orders.order_num = items.order_num
        AND items.total_price > 100.00
```

If you must provide names for some of the columns in a view, then you must provide names for all the columns; that is, the column list must contain an entry for every column that appears in the view.

Using a View in the SELECT Statement

You can define a view in terms of other views, but you must abide by the restrictions on creating views that are listed in Chapter 10 of the *Informix Guide to SQL: Tutorial*. See that manual for further information.

WITH CHECK OPTION Keywords

The WITH CHECK OPTION keywords instruct the database server to ensure that all modifications that are made through the view to the underlying tables satisfy the definition of the view.

The following example creates a view that is named **palo_alto**, which uses all the information in the **customer** table for customers in the city of Palo Alto. The database server checks any modifications made to the **customer** table through **palo_alto** because the WITH CHECK OPTION is specified.

```
CREATE VIEW palo_alto AS
    SELECT * FROM customer
        WHERE city = 'Palo Alto'
        WITH CHECK OPTION
```

What do the WITH CHECK OPTION keywords really check and prevent? It is possible to insert into a view a row that does not satisfy the conditions of the view (that is, a row that is not visible through the view). It is also possible to update a row of a view so that it no longer satisfies the conditions of the view. For example, if the view was created without the WITH CHECK OPTION keywords, you could insert a row through the view where the city is Los Altos, or you could update a row through the view by changing the city from Palo Alto to Los Altos.

To prevent such inserts and updates, you can add the WITH CHECK OPTION keywords when you create the view. These keywords ask the database server to test every inserted or updated row to ensure that it meets the conditions that are set by the WHERE clause of the view. The database server rejects the operation with an error if the row does not meet the conditions.

However, even if the view was created with the WITH CHECK OPTION keywords, you can perform inserts and updates through the view to change columns that are not part of the view definition. A column is not part of the view definition if it does not appear in the WHERE clause of the SELECT statement that defines the view.

Updating Through Views

If a view is built on a single table, the view is *updatable* if the SELECT statement that defined it did not contain any of the following items:

- Columns in the select list that are aggregate values
- Columns in the select list that use the UNIQUE or DISTINCT keyword
- A GROUP BY clause
- A derived value for a column, which was created using an arithmetical expression

In an updatable view, you can update the values in the underlying table by inserting values into the view.

Important: *You cannot update or insert rows in a remote table through views with check options.*

References

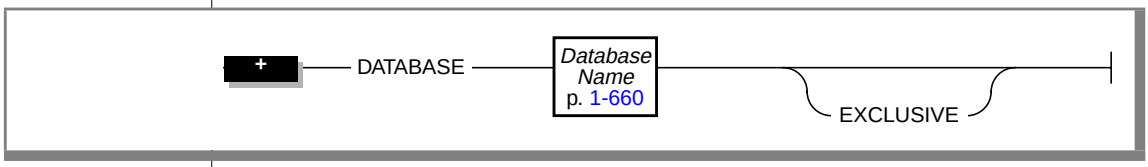
See the CREATE TABLE, DROP VIEW, GRANT, SELECT, and SET SESSION AUTHORIZATION statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of views and security in Chapter 10.

DATABASE

Use the DATABASE statement to select an accessible database as the current database.

Syntax



Usage

You can use the DATABASE statement to select any database on your database server. To select a database on another OnLine database server, specify the name of the database server with the database name.

If you specify the name of the current database server or another database server with the database name, the database server name cannot be uppercase.

Issuing a DATABASE statement when a database is already open closes the current database before opening the new one. Closing the current database releases any cursor resources held by the database server, which invalidates any cursors you have declared up to that point. If the user identity was changed through a SET SESSION AUTHORIZATION statement, the original user name is restored.

The current user (or PUBLIC) must have the Connect privilege on the database specified in the DATABASE statement. The current user cannot have the same user name as an existing role in the database.

ESQL

You cannot include the DATABASE statement in a multistatement PREPARE operation.

You can determine the type of database a user selects by checking the warning flag after a DATABASE statement in the **sqlca** structure.

If the database has transactions, the second element of the **sqlwarn** structure contains a W after the DATABASE statement executes. See the following table for the name of the variable that each SQL API product uses.

Product	Field Name
ESQL / C	sqlca.sqlwarn.sqlwarn1
ESQL / COBOL	SQLWARN1 OF SQLWARN OF SQLCA

♦

ESQL
ANSI

If the database is ANSI compliant, the third element of the **sqlwarn** structure contains a W after the DATABASE statement executes. See the following table for the name of the variable that each product uses.

Product	Field Name
ESQL / C	sqlca.sqlwarn.sqlwarn2
ESQL / COBOL	SQLWARN2 OF SQLWARN OF SQLCA

♦

ESQL

If the database is an INFORMIX-OnLine Dynamic Server database, the fourth element of the **sqlwarn** structure contains a W after the DATABASE statement executes. See the following table for the name of the variable that each product uses.

Product	Field Name
ESQL / C	sqlca.sqlwarn.sqlwarn3
ESQL / COBOL	SQLWARN3 OF SQLWARN OF SQLCA

If the database is running in secondary mode, the seventh element of the **sqlwarn** structure contains a W after the DATABASE statement executes. See the following table for the name of the variable that each product uses.

Product	Field Name
ESQL/C	sqlca.sqlwarn.sqlwarn6
ESQL/COBOL	SQLWARN6 OF SQLWARN OF SQLCA

♦

SE

Only the databases stored in your current directory, or in a directory specified in your **DBPATH** environment variable, are recognized. ♦

SE

ESQL

To specify a database that does not reside in your current directory or in a directory specified by the **DBPATH** environment variable, follow the DATABASE keyword with a program or host variable that evaluates to the full pathname of the database (excluding the **.dbs** extension). ♦

EXCLUSIVE Keyword

The EXCLUSIVE keyword opens the database in exclusive mode and prevents access by anyone but the current user. To allow others access to the database, you must execute the CLOSE DATABASE statement and then reopen the database without the EXCLUSIVE keyword.

The following statement opens the **stores7** database on the **training** database server in exclusive mode:

```
DATABASE stores7@training EXCLUSIVE
```

If another user has already opened the database, exclusive access is denied, an error is returned, and no database is opened.

References

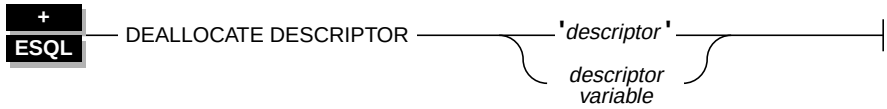
See the CLOSE DATABASE and CONNECT statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of database design in Chapter 8 and implementing the data model in Chapter 9.

DEALLOCATE DESCRIPTOR

Use the DEALLOCATE DESCRIPTOR statement to free a system-descriptor area that was previously allocated for a specified *descriptor* or *descriptor variable*.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated. The surrounding quotes must be single.	Quoted String, p. 1-757
descriptor variable	Host variable name that identifies a system-descriptor area	System-descriptor area must already be allocated.	Variable name must conform to language-specific rules for variable names.

Usage

The DEALLOCATE DESCRIPTOR statement frees all the memory that is associated with the system-descriptor area that *descriptor* or *descriptor variable* identifies. It also frees all the value descriptors (including memory for data values in the value descriptors).

You can reuse a descriptor or descriptor variable after it is deallocated. Deallocation occurs automatically at the end of the program.

If you deallocate a nonexistent descriptor or descriptor variable, an error results.

E/C

You cannot use the DEALLOCATE DESCRIPTOR statement to deallocate an **sqlda** structure. You can use it only to free the memory that is allocated for a system-descriptor area. ♦

The following examples show the DEALLOCATE DESCRIPTOR statement for INFORMIX-ESQL / C and INFORMIX-ESQL / COBOL, respectively. In each example, the first line shows an embedded-variable name, and the second line shows a quoted string that identifies the allocated system-descriptor area.

INFORMIX-ESQL/C

```
EXEC SQL deallocate descriptor :descname;

EXEC SQL deallocate descriptor 'desc1';
```

INFORMIX-ESQL/COBOL

```
EXEC SQL DEALLOCATE DESCRIPTOR :DESCNAME END-EXEC.

EXEC SQL DEALLOCATE DESCRIPTOR 'DESC1' END-EXEC.
```

References

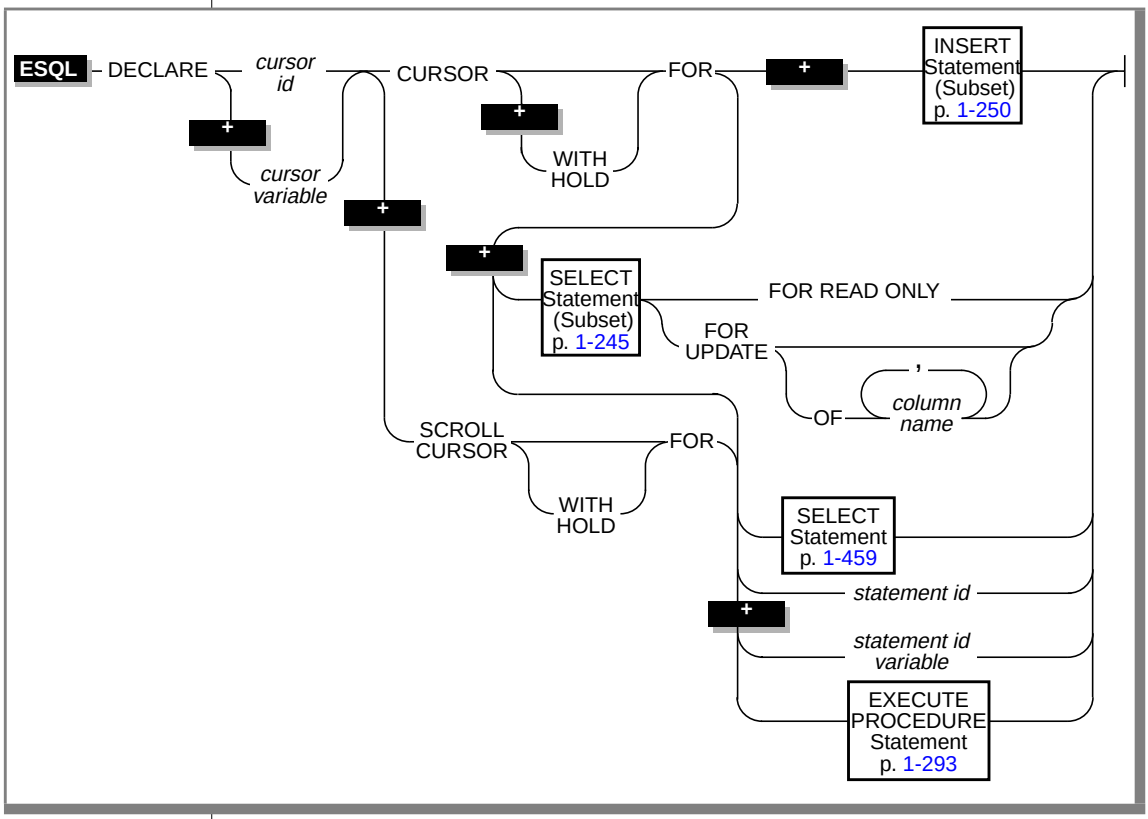
See the ALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of dynamic SQL in Chapter 5.

DECLARE

Use the DECLARE statement to define a cursor that represents the active set of rows that a SELECT, INSERT, or EXECUTE PROCEDURE statement specifies.

Syntax



Element	Purpose	Restrictions	Syntax
column name	A column that you can update through the cursor	The specified column must exist, but it does not have to be in the select list of the SELECT clause.	Identifier, p. 1-723
cursor id	The name that the DECLARE statement assigns to the cursor and that refers to the cursor in other statements	You cannot specify a cursor name that a previous DECLARE statement in the same program has specified.	Identifier, p. 1-723
cursor variable	An embedded variable name that holds the value of *cursor id*	Variable must be a character data type.	The name must conform to language-specific rules for variable names.
statement id	A statement identifier that is a data structure representing the text of a prepared statement	The *statement id* must have already been specified in a PREPARE statement in the same program.	Identifier, p. 1-723, and PREPARE, p. 1-402
statement id variable	An embedded variable name that holds the value of *statement id*	Variable must be a character data type.	The name must conform to language-specific rules for variable names.

Usage

The DECLARE statement associates the cursor with a SELECT, INSERT, or EXECUTE PROCEDURE statement or with the statement identifier (*statement id* or *statement id variable*) of a prepared statement.

The DECLARE statement assigns an identifier to the cursor, specifies its uses, and directs the preprocessor to allocate storage to hold the cursor.

The DECLARE statement must precede any other statement that refers to the cursor during the execution of the program.

When the cursor is used with a SELECT statement, it is a data structure that represents a specific location within the active set of rows that the SELECT statement retrieved. You associate a cursor with an INSERT statement if you want to add multiple rows to the database in an INSERT operation. When the cursor is used with an INSERT statement, it represents the rows that the INSERT statement is to add to the database. When the cursor is used with an EXECUTE PROCEDURE statement, it represents the columns or values that the stored procedure retrieved.

The amount of available memory in the system limits the number of open cursors and prepared statements that you can have at one time in one process. Use FREE *statement id* or FREE *statement id variable* to release the resources that a prepared statement holds; use FREE *cursor id* or FREE *cursor variable* to release resources that a cursor holds.

A program can consist of one or more source-code files. By default, the scope of a cursor is global to a program, so a cursor declared in one file can be referenced from another file.

In a multiple-file program, if you want to limit the scope of cursors to the files in which they are declared, you must preprocess all the files with the **-local** command-line option. See your SQL API product manual for more information, restrictions, and performance issues when you preprocess with the **-local** option.

E/C

A variable used in place of the cursor name or statement identifier must be the CHARACTER data type. In ESQL/C programs, the variable must be defined as `exec sql char`. ♦

E/CO

A variable that is used in place of the cursor name or statement identifier must be the CHARACTER data type. In ESQL/COBOL programs, declare such a variable as a standard CHARACTER type. ♦

To declare multiple cursors, use a single statement identifier. For instance, the following INFORMIX-ESQL/C example does not return an error:

```
EXEC SQL prepare id1 from 'select * from customer';
EXEC SQL declare x cursor for id1;
EXEC SQL declare y scroll cursor for id1;
EXEC SQL declare z cursor with hold for id1;
```

If you include the **-ansi** compilation flag (or if **DBANSIWARN** is set), warnings are generated for statements that use dynamic cursor names or dynamic statement identifier names. Some error checking is performed at runtime. The following list indicates the typical checks:

- Illegal use of cursors (that is, normal cursors used as scroll cursors)
- Use of undeclared cursors
- Bad cursor or statement names (empty)

Checks for multiple declarations of a cursor of the same name are performed at compile time only if the cursor or statement is an identifier. For example, the code in the first example below results in a compile error. The code in the second example does not result in a compile error because it uses a host variable to hold the cursor name.

Results in error

```
EXEC SQL declare x cursor for
    select * from customer;
. . .
EXEC SQL declare x cursor for
    select * from orders;
```

Runs successfully

```
EXEC SQL declare x cursor for
    select * from customer;
. . .
stcopy("x", s);
EXEC SQL declare :s cursor for
    select * from customer;
```

Overview of Cursor Types

Functionally, a cursor can be associated with a SELECT statement (a *select cursor*), an EXECUTE PROCEDURE statement (a *procedure cursor*) that returns values, or an INSERT statement (an *insert cursor*). You can use a select cursor to update or delete rows; it is called an *update cursor.*

A cursor can also be associated with a statement identifier, enabling you to use a cursor with INSERT, SELECT, or EXECUTE PROCEDURE statements that are prepared dynamically and to use different statements with the same cursor at different times. In this case, the type of cursor depends on the statement that is prepared at the time the cursor is opened (see the OPEN statement on page 1-390).

Tip: *Cursors for stored procedures behave the same as select cursors, which are enabled as update cursors.*

Select or Procedure Cursor

A select or procedure cursor enables you to scan multiple rows of data and to move data row by row into a set of receiving variables, as the following steps describe:

1. Use a DECLARE statement to define a cursor for the SELECT statement or for the EXECUTE PROCEDURE statement.
2. Open the cursor with the OPEN statement. The database server processes the query until it locates or constructs the first row of the active set.
3. Retrieve successive rows of data with the FETCH statement.
4. Close the cursor with the CLOSE statement when the active set is no longer needed.
5. Free the cursor with the FREE statement. The FREE statement releases the resources that are allocated for a declared cursor.

A select cursor can be explicitly declared as read only with the FOR READ ONLY option.

Read-Only Cursor

Use the FOR READ ONLY option to state explicitly that a select cursor cannot be used to modify data. In a database that is not ANSI compliant, a select cursor and a select cursor that is built with the FOR READ ONLY option are the same. Neither can be used to update data.

ANSI

In an ANSI-compliant database, if you want a select cursor to be read only, you must use the FOR READ ONLY keywords when you declare the cursor. ♦

Update Cursor

Use the FOR UPDATE keywords to declare an update cursor. You can use the update cursor to modify (update or delete) the current row.

ANSI

In an ANSI-compliant database, you can use a select cursor to update or delete data as long as the cursor was not declared with the FOR READ ONLY keywords and it follows the restrictions on update cursors that are described in "Subset of the SELECT Statement Associated with Cursors" on page 1-245. You do not need to use the FOR UPDATE keywords when you declare the cursor. ♦

Insert Cursor

An insert cursor increases processing efficiency (compared with embedding the INSERT statement directly). The insert cursor allows bulk insert data to be buffered in memory and written to disk when the buffer is full. This process reduces communication between the program and the database server and also increases the speed of the insertions.

Cursor Characteristics

Structurally, you can declare a cursor as a *sequential* cursor (the default condition), a *scroll* cursor (using the SCROLL keyword), or a *hold* cursor (using the WITH HOLD keywords). The following sections explain these structural characteristics.

Sequential Cursor

If you use only the CURSOR keyword in a DECLARE statement, you create a sequential cursor, which can fetch only the next row in sequence from the active set. The sequential cursor can read through the active set only once each time it is opened. If you are using a sequential cursor, on each execution of the FETCH statement, the database server returns the contents of the current row and locates the next row in the active set.

The following INFORMIX-ESQL / C example is read only in a database that is not ANSI compliant and read / updatable in an ANSI-compliant database:

```
EXEC SQL declare s_cur cursor for
    select fname, lname into :st_fname, :st_lname
    from orders where customer_num = 114;
```

Scroll Cursor

The SCROLL keyword creates a scroll cursor, which you can use to fetch rows of the active set in any sequence. To implement a scroll cursor, the database server creates a temporary table to hold the active set. With the active set retained as a table, you can fetch the first, last, or any intermediate rows as well as fetch rows repeatedly without having to close and reopen the cursor. See the FETCH statement on page 1-296 for a discussion of these abilities.

The database server retains the active set for a scroll cursor in a temporary table until the cursor is closed. On a multiuser system, the rows in the tables from which the active-set rows were derived might change after a copy is made in the temporary table. (For information about temporary tables, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.) If you use a scroll cursor within a transaction, you can prevent copied rows from changing either by setting the isolation level to Repeatable Read (available only with INFORMIX-OnLine Dynamic Server) or by locking the entire table in share mode during the transaction. (See the SET ISOLATION statement on page 1-575 and the LOCK TABLE statement on page 1-387.)

The following example creates a scroll cursor:

```
DECLARE sc_cur SCROLL CURSOR FOR
    SELECT * FROM orders
```

Hold Cursor

If you use the WITH HOLD keywords, you create a hold cursor. A hold cursor remains open after a transaction ends. You can use the WITH HOLD keywords to declare both sequential and scroll cursors. The following example creates a hold cursor:

```
DECLARE hld_cur CURSOR WITH HOLD FOR
    SELECT customer_num, lname, city FROM customer
```

A hold cursor allows uninterrupted access to a set of rows across multiple transactions. Ordinarily, all cursors close at the end of a transaction; a hold cursor does not close.

You can use a hold cursor as the following ESQL / C code example shows. This code fragment uses a hold cursor as a *master* cursor to scan one set of records and a sequential cursor as a *detail* cursor to point to records that are located in a different table. The records that the master cursor scans are the basis for updating the records to which the detail cursor points. The COMMIT WORK statement at the end of each iteration of the first WHILE loop leaves the hold cursor **c_master** open but closes the sequential cursor **c_detail** and releases all locks. This technique minimizes the resources that the database server must devote to locks and unfinished transactions, and it gives other users immediate access to updated rows.

```
EXEC SQL BEGIN DECLARE SECTION;
int p_custnum,
int save_status;
long p_orddate;
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare st_1 from
    'select order_date
        from orders where customer_num = ? for update';
EXEC SQL declare c_detail cursor for st_1;

EXEC SQL declare c_master cursor with hold for
    select customer_num
        from customer where city = 'Pittsburgh';

EXEC SQL open c_master;
if(SQLCODE==0) /* the open worked */
    EXEC SQL fetch c_master into :p_custnum; /* discover first customer */
while(SQLCODE==0) /* while no errors and not end of pittsburgh customers */
    {
    EXEC SQL begin work; /* start transaction for customer p_custnum */
    EXEC SQL open c_detail using :p_custnum;
    if(SQLCODE==0) /* detail open succeeded */
        EXEC SQL fetch c_detail into :p_orddate; /* get first order */
    while(SQLCODE==0) /* while no errors and not end of orders */
         {
        EXEC SQL update orders set order_date = '08/15/94'
            where current of c_detail;
        if(status==0) /* update was ok */
            EXEC SQL fetch c_detail into :p_orddate; /* next order */
        }
    if(SQLCODE==SQLNOTFOUND) /* correctly updated all found orders */
        EXEC SQL commit work; /* make updates permanent, set status */
    else /* some failure in an update */
         {
        save_status = SQLCODE; /* save error for loop control */
        EXEC SQL rollback work;
        SQLCODE = save_status; /* force loop to end */
         }
    if(SQLCODE==0) /* all updates, and the commit, worked ok */
        EXEC SQL fetch c_master into :p_custnum; /* next customer? */
    }
EXEC SQL close c_master;
```

Use either the CLOSE statement to close the hold cursor explicitly or the CLOSE DATABASE or DISCONNECT statements to close it implicitly. The CLOSE DATABASE statement closes all cursors.

Declaring a Cursor as an Update or Read-Only Cursor

When you associate a cursor with a SELECT statement, you can define it as an update cursor or as a read-only cursor, as follows:

- Use the FOR UPDATE keywords to define the cursor as an update cursor.
- Use the FOR READ ONLY keywords to define the cursor as a read-only cursor.

You cannot specify both the FOR UPDATE option and the FOR READ ONLY option in the same DECLARE statement because these options are mutually exclusive.

Defining an Update Cursor

Use the FOR UPDATE keywords to notify the database server that updating is possible and cause it to use more stringent locking than with a select cursor. You can specify particular columns that can be updated.

After you create an update cursor, you can update or delete the currently selected row by using an UPDATE or DELETE statement with the WHERE CURRENT OF clause. The words CURRENT OF refer to the row that was most recently fetched; they take the place of the usual test expressions in the WHERE clause.

An update cursor lets you perform updates that are not possible with the UPDATE statement because the decision to update and the values of the new data items can be based on the original contents of the row. Your program can evaluate or manipulate the selected data before it decides whether to update. The UPDATE statement cannot interrogate the table that is being updated.

ANSI

All simple select cursors are potentially update cursors even if they are declared without the FOR UPDATE keywords. (See the restrictions on SELECT statements associated with update cursors in "Subset of the SELECT Statement Associated with Cursors" on page 1-245.) ♦

Locking with an Update Cursor

You declare an update cursor to let the database server know that the program might update (or delete) any row that it fetches as part of the SELECT statement. The update cursor employs *promotable* locks for rows that the program fetches. Other programs can read the locked row, but no other program can place a promotable or write lock. Before the program modifies the row, the row lock is promoted to an exclusive lock.

SE

The INFORMIX-SE database server does not use promotable locks. Before the program modifies a row, the database server obtains an exclusive lock on the row. ♦

Although it is possible to declare an update cursor with the WITH HOLD keywords, the only reason to do so is to break a long series of updates into smaller transactions. You must fetch and update a particular row in the same transaction.

If an operation involves fetching and updating a very large number of rows, the lock table that the database server maintains can overflow. The usual way to prevent this overflow is to lock the entire table that is being updated. If this action is impossible, an alternative is to update through a hold cursor and to execute COMMIT WORK at frequent intervals. However, you must plan such an application very carefully because COMMIT WORK releases all locks, even those that are placed through a hold cursor.

Using FOR UPDATE with a List of Columns

When you declare an update cursor, you can limit the update to specific columns by including the OF keyword and a list of columns. You can modify only those named columns in subsequent UPDATE statements. The columns need not be in the select list of the SELECT clause.

This column restriction applies only to UPDATE statements. The OF *column* clause has no effect on subsequent DELETE statements that use a WHERE CURRENT OF clause. (A DELETE statement removes the contents of all columns.)

The principal advantage to specifying columns is documentation and preventing programming errors. (The database server refuses to update any other columns.) An additional advantage is speed, when the SELECT statement meets the following criteria:

- The SELECT statement can be processed using an index.
- The columns that are listed are not part of the index that is used to process the SELECT statement.

If the columns that you intend to update are part of the index that is used to process the SELECT statement, the database server must keep a list of each row that is updated to ensure that no row is updated twice. When you use the OF keyword to specify the columns that can be updated, the database server determines whether to keep the list of updated rows. If the database server determines that the list is unnecessary, then eliminating the work of keeping the list results in a performance benefit. If you do not use the OF keyword, the database server keeps the list of updated rows, although it might be unnecessary.

The following example contains INFORMIX-ESQL / C code that uses an update cursor with a DELETE statement to delete the current row. Whenever the row is deleted, the cursor remains between rows. After you delete data, you must use a FETCH statement to advance the cursor to the next row before you can refer to the cursor in a DELETE or UPDATE statement.

```
EXEC SQL declare q_curs cursor for
    select * from customer where lname matches :last_name
        for update;

EXEC SQL open q_curs;
for (;;)
{
    EXEC SQL fetch q_curs into :cust_rec;
    if (strncmp(SQLSTATE, "00", 2) != 0)
        break;

    /* Display customer values and prompt for answer */
    printf("\n%s %s", cust_rec.fname, cust_rec.lname);
    printf("\nDelete this customer? ");
    scanf("%s", answer);

    if (answer[0] == 'y')
        EXEC SQL delete from customer where current of q_curs;
    if (strncmp(SQLSTATE, "00", 2) != 0)
        break;
}
printf("\n");
EXEC SQL close q_curs;
```

Defining a Read-Only Cursor

Use the FOR READ ONLY keywords to define a cursor as a read-only cursor. The need for the FOR READ ONLY keywords depends on whether your database is an ANSI-mode database or a database that is not ANSI compliant.

In a database that is not ANSI compliant, the cursor that the DECLARE statement defines is a read-only cursor by default. So you do not need to specify the FOR READ ONLY keywords if you want the cursor to be a read-only cursor. The only advantage of specifying the FOR READ ONLY keywords explicitly is for better program documentation.

ANSI

In an ANSI-mode database, the cursor associated with a SELECT statement through the DECLARE statement is an update cursor by default, provided that the SELECT statement conforms to all of the restrictions for update cursors listed in "Subset of the SELECT Statement Associated with Cursors" below.

Therefore, you should use the FOR READ ONLY keywords in the DECLARE statement only if you want the cursor to be a read-only cursor rather than an update cursor. You declare a read-only cursor to let the database server know that the program will not update (or delete) any row that it fetches as part of the SELECT statement. The database server can use less stringent locking for a read-only cursor than for an update cursor. ♦

Subset of the SELECT Statement Associated with Cursors

Not all SELECT statements can be associated with an update cursor or a read-only cursor. If the DECLARE statement includes the FOR UPDATE clause or the FOR READ ONLY clause, you must observe certain restrictions on the SELECT statement that is included in the DECLARE statement (either directly or as a prepared statement).

If the DECLARE statement includes the FOR UPDATE clause, the SELECT statement must conform to the following restrictions:

- The statement can select data from only one table.
- The statement cannot include any aggregate functions.
- The statement cannot include any of the following clauses or keywords: DISTINCT, FOR READ ONLY, FOR UPDATE, GROUP BY, INTO TEMP, ORDER BY, UNION, or UNIQUE.

If the DECLARE statement includes the FOR READ ONLY clause, the SELECT statement must conform to the following restrictions:

- The SELECT statement cannot have a FOR READ ONLY clause.
- The SELECT statement cannot have a FOR UPDATE clause.

For a complete description of SELECT syntax and usage, see the SELECT statement on page 1-459.

Examples of Cursors in ANSI and non-ANSI Databases

In a database that is not ANSI compliant, a cursor associated with a SELECT statement is a read-only cursor by default. The following example declares a read-only cursor in a non-ANSI database:

```
EXEC SQL declare cust_curs cursor for
    select * from customer_notansi;
```

If you want to make it clear in the program code that this cursor is a read-only cursor, you can specify the FOR READ ONLY option as shown in the following example:

```
EXEC SQL declare cust_curs cursor for
    select * from customer_notansi
    for read only;
```

If you want this cursor to be an update cursor, you need to specify the FOR UPDATE option in your DECLARE statement. The following example declares an update cursor:

```
EXEC SQL declare new_curs cursor for
    select * from customer_notansi
    for update;
```

If you want an update cursor to be able to modify only some of the columns in a table, you need to specify these columns in the FOR UPDATE option.The following example declares an update cursor and specifies that this cursor can update only the **fname** and **lname** columns in the **customer_notansi** table:

```
EXEC SQL declare name_curs cursor for
    select * from customer_notansi
    for update of fname, lname;
```

ANSI

In an ANSI-mode database, a cursor associated with a SELECT statement is an update cursor by default. The following example declares an update cursor in an ANSI-mode database:

```
EXEC SQL declare x_curs cursor for
    select * from customer_ansi;
```

If you want to make it clear in the program documentation that this cursor is an update cursor, you can specify the FOR UPDATE option as shown in the following example:

```
EXEC SQL declare x_curs cursor for
    select * from customer_ansi
    for update;
```

If you want an update cursor to be able to modify only some of the columns in a table, you must specify these columns in the FOR UPDATE option. The following example declares an update cursor and specifies that this cursor can update only the **fname** and **lname** columns in the **customer_ansi** table:

```
EXEC SQL declare y_curs cursor for
    select * from customer_ansi
    for update of fname, lname;
```

If you want a cursor to be a read-only cursor, you must override the default behavior of the DECLARE statement by specifying the FOR READ ONLY option in your DECLARE statement. The following example declares a read-only cursor:

```
EXEC SQL declare z_curs cursor for
    select * from customer_ansi
    for read only;
```

♦

Associating a Cursor with a Prepared Statement

The PREPARE statement lets you assemble the text of an SQL statement at runtime and pass the statement text to the database server for execution. If you anticipate that a dynamically prepared SELECT statement or EXECUTE PROCEDURE statement that returns values could produce more than one row of data, the prepared statement must be associated with a cursor. (See the PREPARE statement on page 1-402 for more information about preparing SQL statements.)

The result of a PREPARE statement is a statement identifier (*statement id* or *id variable*), which is a data structure that represents the prepared statement text. You declare a cursor for the statement text by associating a cursor with the statement identifier.

You can associate a sequential cursor with any prepared SELECT or EXECUTE PROCEDURE statement. You cannot associate a scroll cursor with a prepared INSERT statement or with a SELECT statement that was prepared to include a FOR UPDATE clause.

After a cursor is opened, used, and closed, a different statement can be prepared under the same statement identifier. In this way, it is possible to use a single cursor with different statements at different times. The cursor must be redeclared before you use it again.

The following example contains INFORMIX-ESQL/C code that prepares a SELECT statement and declares a cursor for the prepared statement text. The statement identifier **st_1** is first prepared from a SELECT statement that returns values; then the cursor **c_detail** is declared for **st_1**.

```
EXEC SQL prepare st_1 from
    'select order_date
        from orders where customer_num = ?';
EXEC SQL declare c_detail cursor for st_1;
```

If you want use a prepared SELECT statement to modify data, add a FOR UPDATE clause to the statement text that you wish to prepare, as the following INFORMIX-ESQL/C example shows:

```
EXEC SQL prepare sel_1 from 'select * from customer for update';
EXEC SQL declare sel_curs cursor for sel_1;
```

Using Cursors with Transactions

To roll back a modification, you must perform the modification within a transaction. A transaction in a database that is not ANSI-compliant begins only when the BEGIN WORK statement is executed.

ANSI

In ANSI-compliant databases, transactions are always in effect. ♦

The database server enforces the following guidelines for select and update cursors. These guidelines ensure that modifications can be committed or rolled back properly:

- Open an insert or update cursor within a transaction.
- Include PUT and FLUSH statements within one transaction.
- Modify data (update, insert, or delete) within one transaction.

The database server lets you open and close a hold cursor for an update outside a transaction; however, you should fetch all the rows that pertain to a given modification and then perform the modification all within a single transaction. You cannot open and close hold or update cursors outside a transaction.

The following example produces an error when the database server tries to execute the UPDATE statement:

Results in error

```
EXEC SQL declare q_curs cursor for
    select customer_num, fname, lname from customer
    where lname matches :last_name
        for update;
EXEC SQL open q_curs;
EXEC SQL fetch q_curs into :cust_rec; /* fetch before begin */
EXEC SQL begin work;
EXEC SQL update customer set lname = 'Smith'
    where current of q_curs;
/* error here */
EXEC SQL commit work;
```

The following example does not produce an error when the database server tries to execute the UPDATE statement:

Runs successfully

```
EXEC SQL declare q_curs cursor for
    select customer_num, fname, lname from customer
    where lname matches :last_name
        for update;
EXEC SQL open q_curs;
EXEC SQL begin work;
EXEC SQL fetch q_curs into :cust_rec; /* fetch after begin */
EXEC SQL update customer set lname = 'Smith'
    where current of q_curs;
/* no error */
EXEC SQL commit work;
```

When you update a row within a transaction, the row remains locked until the cursor is closed or the transaction is committed or rolled back. If you update a row when no transaction is in effect, the row lock is released when the modified row is written to disk.

If you update or delete a row outside a transaction, you cannot roll back the operation.

A cursor that is declared for insert is an insert cursor. In a database that uses transactions, you cannot open an insert cursor outside a transaction unless it was also declared with hold.

Subset of INSERT Associated with a Sequential Cursor

To create an insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. The INSERT statement must include a VALUES clause; it cannot contain an embedded SELECT statement.

The following example contains INFORMIX-ESQL/C code that declares an insert cursor:

```
EXEC SQL declare ins_cur cursor for
    insert into stock values
    (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

The insert cursor simply inserts rows of data; it cannot be used to fetch data. When an insert cursor is opened, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program executes PUT statements. The rows are written to disk only when the buffer is full. You can use the CLOSE, FLUSH, or COMMIT WORK statement to flush the buffer when it is less than full. This topic is discussed further under the PUT and CLOSE statements. You must close an insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly.

Using an Insert Cursor with Hold

If you associate a hold cursor with an INSERT statement, you can use transactions to break a long series of PUT statements into smaller sets of PUT statements. Instead of waiting for the PUT statements to fill the buffer and trigger an automatic write to the database, you can execute a COMMIT WORK statement to flush the row buffer. If you use a hold cursor, the COMMIT WORK statement commits the inserted rows but leaves the cursor open for further inserts. This method can be desirable when you are inserting a large number of rows, because pending uncommitted work consumes database server resources.

References

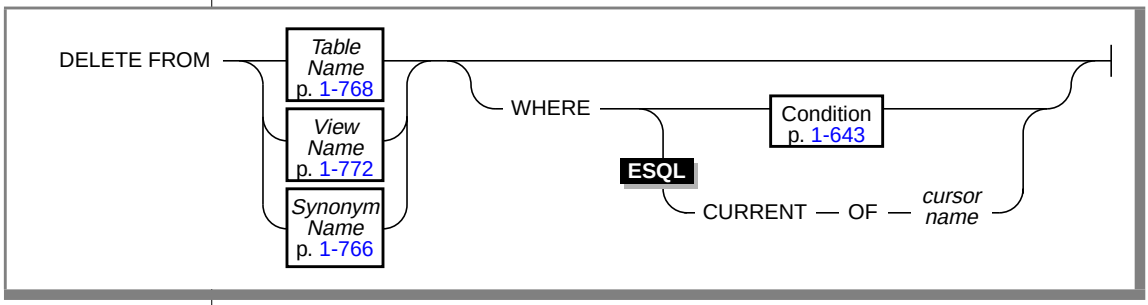
See the CLOSE, DELETE, EXECUTE PROCEDURE, FETCH, FREE, INSERT, OPEN, PREPARE, PUT, SELECT, and UPDATE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of cursors and data modification in Chapter 5 and Chapter 6, respectively.

DELETE

Use the DELETE statement to delete one or more rows from a table.

Syntax



Element	Purpose	Restrictions	Syntax
cursor name	The name of the cursor whose current row is to be deleted	The cursor must have been previously declared in a DECLARE statement with a FOR UPDATE clause.	Identifier, p. 1-723

Usage

If you use the DELETE statement without a WHERE clause, all the rows in the table are deleted.

If you use the DELETE statement outside a transaction in a database that uses transactions, each DELETE statement that you execute is treated as a single transaction.

Each row affected by a DELETE statement within a transaction is locked for the duration of the transaction; therefore, a single DELETE statement that affects a large number of rows locks the rows until the entire operation is complete. If the number of rows affected is very large, you might exceed the limits your operating system places on the maximum number of simultaneous locks. If this occurs, you can either reduce the scope of the DELETE statement or lock the entire table before you execute the statement.

If you specify a view name, the view must be updatable. See "Updating Through Views" on page 1-228 for an explanation of an updatable view.

DB

If you omit the WHERE clause while you are working within the SQL menu, DB-Access prompts you to verify that you want to delete all rows from a table. You do not receive a prompt if you run the DELETE statement within a command file. ♦

ANSI

Statements are always within an implicit transaction in an ANSI-compliant database; therefore, you cannot have a DELETE statement outside a transaction. ♦

Using Cascading Deletes

Use the ON DELETE CASCADE option of the REFERENCES clause on either the CREATE TABLE or ALTER TABLE statement to specify that you want deletes to cascade from one table to another. For example, the **stock** table contains the column **stock_num** as a primary key. The **catalog** and **items** tables each contain the column **stock_num** as foreign keys with the ON DELETE CASCADE option specified. When a delete is performed from the **stock** table, rows are also deleted in the **catalog** and **items** tables, which are referred through the foreign keys.

If a cascading delete is performed without a WHERE clause, all rows in the parent table (and subsequently, the affected child tables) are deleted.

WHERE Clause

Use the WHERE clause to specify one or more rows that you want to delete. The WHERE conditions are the same as the conditions in the SELECT statement. For example, the following statement deletes all the rows of the **items** table where the order number is less than 1034:

```
DELETE FROM items
    WHERE order_num < 1034
```

DB

If you include a WHERE clause that selects all rows in the table, DB-Access gives no prompt and deletes all rows. ♦

Deleting and the WHERE Clause

ANSI

If you delete from a table in an ANSI-compliant database using a WHERE clause and no rows are found, you can detect this condition using the GET DIAGNOSTICS statement. The **RETURNED_SQLSTATE** field of the GET DIAGNOSTICS statement contains the value '02000.' In a database that is not ANSI compliant, no error is returned. ♦

If you delete from a table using a WHERE clause in a multistatement prepare in either ANSI databases and databases that are not ANSI-compliant and no rows are found, you receive a **RETURNED_SQLSTATE** field value of '02000.'

For additional information about the SQLSTATE code, see the GET DIAGNOSTICS statement in this manual.

You can also use the SQLCODE field of **sqlca** to determine the same results. See the *Informix Guide to SQL: Tutorial* for further information about the SQLCODE field of **sqlca.**

CURRENT OF Clause

ESQL

To use the CURRENT OF clause, you must have previously used the DECLARE statement with the FOR UPDATE clause to announce the *cursor name*.

If you use the CURRENT OF clause, the DELETE statement removes the row of the active set at the current position of the cursor. After the deletion, no current row exists; you cannot use the cursor to delete or update a row until you reposition the cursor with a FETCH statement. ♦

ANSI
ESQL

All select cursors are potentially update cursors in ANSI-compliant databases. You can use the CURRENT OF clause with any select cursor. ♦

References

See the INSERT, UPDATE, DECLARE, GET DIAGNOSTICS, and FETCH statements in this manual.

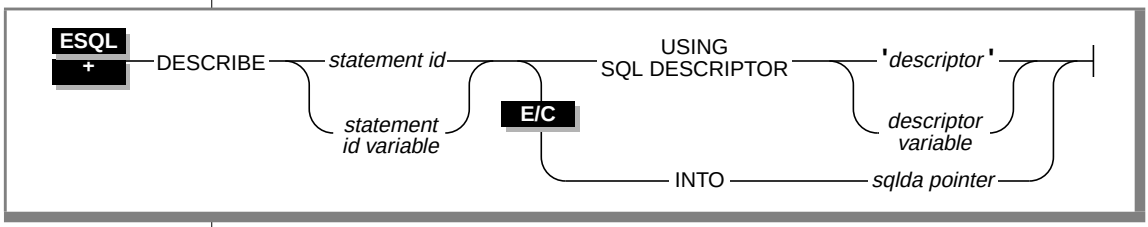
In the *Informix Guide to SQL: Tutorial*, see the discussions of cursors and data modification in Chapter 5 and Chapter 6, respectively.

In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the DELETE statement.

DESCRIBE

Use the DESCRIBE statement to obtain information about a prepared statement before you execute it. The DESCRIBE statement returns the prepared statement type. For a SELECT, EXECUTE PROCEDURE, or INSERT statement, the DESCRIBE statement also returns the number, data types and size of the values, and the name of the column or expression that the query returns. The information can be stored in a system-descriptor area or, in ESQL/C, in an **sqlda** structure.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-757
descriptor variable	Host variable name that identifies a system-descriptor area	Variable must contain the name of an allocated system-descriptor area.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
sqlda pointer	A pointer to an **sqlda** structure	You cannot begin an sqlda pointer with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	See the discussion of **sqlda** structure in the *INFORMIX-ESQL/C Programmer's Manual*.
statement id	Statement identifier for a prepared SQL statement	The statement identifier must be defined in a previous PREPARE statement.	PREPARE, p. 1-402
statement id variable	Host variable that contains a statement identifier for a prepared SQL statement	The statement identifier must be defined in a previous PREPARE statement. The variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

Usage

The DESCRIBE statement allows you to determine, at runtime, the type of statement that has been prepared and the number and types of data that a prepared query returns when it is executed. With this information, you can write code to allocate memory to hold retrieved values and display or process them after they are fetched.

Describing the Statement Type

The DESCRIBE statement takes a statement identifier from a PREPARE statement as input. When the DESCRIBE statement executes, the database server sets the value of the SQLCODE field of the **sqlca** (see the manual for your SQL API product) to indicate the statement type (that is, the keyword with which the statement begins). If the prepared statement text contains more than one SQL statement, the DESCRIBE statement returns the type of the first statement in the text.

SQLCODE is set to zero to indicate a SELECT statement *without* an INTO TEMP clause. This situation is the most common. For any other SQL statement, SQLCODE is set to a positive integer. See the manual for your SQL API product for more information about possible SQLCODE values after a DESCRIBE statement.

You can test the number against the constant names that are defined. In INFORMIX-ESQL/C, the constant names are defined in the **sqlstype.h** header file. A printed list of the possible values and their constant names appears in the manual for each SQL API product.

The DESCRIBE statement uses the SQLCODE field differently than any other statement, possibly returning a nonzero value when it executes successfully. You can revise standard error-checking routines to accommodate this behavior, if desired.

Checking for Existence of a WHERE Clause

If the DESCRIBE statement detects that a prepared statement contains an UPDATE or DELETE statement without a WHERE clause, the DESCRIBE statement sets the following **sqlca** variable to W.

Product	Field Name
ESQL/C	sqlca.sqlwarn.sqlwarn4
ESQL/COBOL	SQLWARN4 OF SQLWARN OF SQLCA

Without a WHERE clause, the update or delete action is applied to the entire table. Check this variable to avoid unintended global changes to your table.

Describing SELECT, EXECUTE PROCEDURE, or INSERT

If the prepared statement text includes a SELECT statement without an INTO TEMP clause, an EXECUTE PROCEDURE statement, or an INSERT statement, the DESCRIBE statement also returns a description of each column or expression that is included in the SELECT, EXECUTE PROCEDURE, or INSERT list. These descriptions are stored in a system-descriptor area or in a pointer to an **sqlda** structure.

The description includes the following information:

- The data type of the column, as defined in the table
- The length of the column, in bytes
- The name of the column or expression

See Chapter 5 of the *Informix Guide to SQL: Tutorial* for more information on the system-descriptor area.

You can modify the system-descriptor-area information and use it in statements that support a USING SQL DESCRIPTOR clause, such as EXECUTE, FETCH, OPEN, and PUT. You must modify the system-descriptor area to show the address in memory that is to receive the described value. You can change the data type to another compatible type. This change causes data conversion to take place when the data is fetched.

In addition to Chapter 5 of the *Informix Guide to SQL: Tutorial*, see the manual for your SQL API for further information about interpreting and using the data that is contained in the system-descriptor area.

USING SQL DESCRIPTOR Clause

The USING SQL DESCRIPTOR clause lets you store the description of a SELECT, INSERT, or EXECUTE PROCEDURE list in a system-descriptor area that an ALLOCATE DESCRIPTOR statement creates. You can obtain information about the resulting columns of a prepared statement through a system-descriptor area. Use the USING SQL DESCRIPTOR keywords and a descriptor to point to a system-descriptor area instead of to an **sqlda** structure.

The DESCRIBE statement sets the COUNT field in the system-descriptor area to the number of values in the SELECT, EXECUTE PROCEDURE, or INSERT list. If COUNT is greater than the number of item descriptors (*occurrences*) in the system-descriptor area, the system returns an error. Otherwise, the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE information is set and memory for DATA fields is allocated automatically.

After a DESCRIBE statement is executed, the SCALE and PRECISION fields contain the scale and precision of the column, respectively. If SCALE and PRECISION are set in the SET DESCRIPTOR statement, and TYPE is set to DECIMAL or MONEY, the LENGTH field is modified to adjust for the scale and precision of the decimal value. If TYPE is not set to DECIMAL or MONEY, the values for SCALE and PRECISION are not set, and LENGTH is unaffected.

The following examples show the use of a system descriptor in a DESCRIBE statement in INFORMIX-ESQL / C and INFORMIX-ESQL / COBOL. In the first example in each pair, the descriptor is a quoted string; in the second example in each pair, it is an embedded variable name.

INFORMIX-ESQL/C

```
main()
{
. . .
EXEC SQL allocate descriptor 'desc1' with max 3;
EXEC SQL prepare curs1 FROM 'select * from tab';
EXEC SQL describe curs1 using sql descriptor 'desc1';
}

EXEC SQL describe curs1 using sql descriptor :desc1var;
```

INFORMIX-ESQL/COBOL

```
EXEC SQL ALLOCATE DESCRIPTOR 'DESC1' WITH MAX 3 END-EXEC.
EXEC SQL PREPARE CURS1 FROM 'SELECT * FROM TAB' END-EXEC.
EXEC SQL DESCRIBE CURS1 USING SQL DESCRIPTOR 'DESC1' END-EXEC.

EXEC SQL DESCRIBE CURS1 USING SQL DESCRIPTOR :DESC1VAR END-EXEC.
```

INTO sqlda pointer Clause

E/C

The INTO *sqlda pointer* clause lets you allocate memory for an **sqlda** structure and store its address in an **sqlda** pointer. The DESCRIBE statement fills in the allocated memory with descriptive information. The DESCRIBE statement sets the **sqlda.sqld** field to the number of values in the SELECT, INSERT, or EXECUTE PROCEDURE list. The **sqlda** structure also contains an array of data descriptors (**sqlvar** structures), one for each value in the SELECT, INSERT, or EXECUTE PROCEDURE list. After a DESCRIBE statement is executed, the **sqlda.sqlvar** structure has the **sqltype**, **sqllen**, and **sqlname** fields set.

The DESCRIBE statement allocates memory for an **sqlda** pointer once it is declared in a program. However, the application program must designate the storage area of the **sqlda.sqlvar.sqldata** fields.

See the *INFORMIX-ESQL/C Programmer's Manual* for further information on the **sqlda** structure. ♦

E/CO

This product does not support pointers to an **sqlda** structure; it returns an error if you try to execute a DESCRIBE statement that uses one. Only system-descriptor areas that are allocated with the ALLOCATE DESCRIPTOR statement can be used in a DESCRIBE statement in INFORMIX-ESQL / COBOL. You can view the contents of the columns by executing a GET DESCRIPTOR statement following a DESCRIBE statement on the specified system descriptor. ♦

References

See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual for further information about using dynamic management statements.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the DESCRIBE statement in Chapter 5.

For further information about how to use a system-descriptor area or an **sqlda** pointer if you intend to use a FETCH...USING DESCRIPTOR or an INSERT...USING DESCRIPTOR statement, refer to the manual for your SQL API product.

DISCONNECT

The DISCONNECT statement terminates a connection between an application and a database server.

Syntax

Element

Usage

The DISCONNECT statement lets you terminate a connection to a database server. If a database is open, it closes before the connection drops. Even if you made a connection to a specific database only, that connection to the database server is terminated by the DISCONNECT statement.

You cannot use the PREPARE statement for the DISCONNECT statement.

ESQL

If you disconnect a specific connection using *connection name* or *conn_nm variable*, DISCONNECT generates an error if the specified connection is not a current or dormant connection.

A DISCONNECT statement that does not terminate the current connection does not change the context of the current environment (the connection context). ♦

The DEFAULT Option

Use the DEFAULT option to identify the default connection for a DISCONNECT statement. The default connection is one of the following connections:

- An explicit default connection (a connection established with the CONNECT TO DEFAULT statement)
- An implicit default connection (any connection made using the DATABASE, CREATE DATABASE, or START DATABASE statements)

You can use DISCONNECT to disconnect the default connection. See "The DEFAULT Option" on page 1-91 and "The Implicit Connection with DATABASE Statements" on page 1-92 for more information.

If the DATABASE statement does not specify a database server, as shown in the following example, the default connection is made to the default database server:

```
EXEC SQL database 'stores7';
.
.
.
EXEC SQL disconnect default;
```

If the DATABASE statement specifies a database server, as shown in the following example, the default connection is made to that database server:

```
EXEC SQL database 'stores7@mydbsrvr';
.
.
.
EXEC SQL disconnect default;
```

In either case, the DEFAULT option of DISCONNECT disconnects this default connection. See "The DEFAULT Option" on page 1-91 and "The Implicit Connection with DATABASE Statements" on page 1-92 for more information about the default database server and implicit connections.

The CURRENT Keyword

Use the CURRENT keyword with the DISCONNECT statement as a shorthand form of identifying the current connection. The CURRENT keyword replaces the current connection name. For example, the DISCONNECT statement in the following excerpt terminates the current connection to the database server **mydbsrvr**:

```
CONNECT TO 'stores7@mydbsrvr'
.
.
.
DISCONNECT CURRENT
```

When a Transaction is Active

When a transaction is active, the DISCONNECT statement generates an error. The transaction remains active, and the application must explicitly commit it or roll it back. If an application terminates without issuing a DISCONNECT statement (because of a system crash or program error, for example), active transactions are rolled back.

Disconnecting in a Thread-Safe Environment

E/C

If you issue the DISCONNECT statement in a thread-safe ESQL / C application, keep in mind that an active connection can only be disconnected from within the thread in which it is active. Therefore one thread cannot disconnect another thread's active connection. The DISCONNECT statement generates an error if such an attempt is made.

However, once a connection becomes dormant, any other thread can disconnect this connection unless an ongoing transaction is associated with the dormant connection (the connection was established with the WITH CONCURRENT TRANSACTION clause of CONNECT). If the dormant connection was not established with the WITH CONCURRENT TRANSACTION clause, DISCONNECT generates an error when it tries to disconnect it.

See the SET CONNECTION statement on page 1-527 for an explanation of connections in a thread-safe ESQL / C application. ♦

Specifying the ALL Option

Use the keyword ALL to terminate all connections established by the application up to that time. For example, the following DISCONNECT statement disconnects the current connection as well as all dormant connections:

```
DISCONNECT ALL
```

E/C

The ALL keyword has the same effect on multithreaded applications that it has on single-threaded applications. Execution of the DISCONNECT ALL statement causes all connections in all threads to be terminated. However, the DISCONNECT ALL statement fails if any of the connections is in use or has an ongoing transaction associated with it. If either of these conditions is true, none of the connections is disconnected. ♦

References

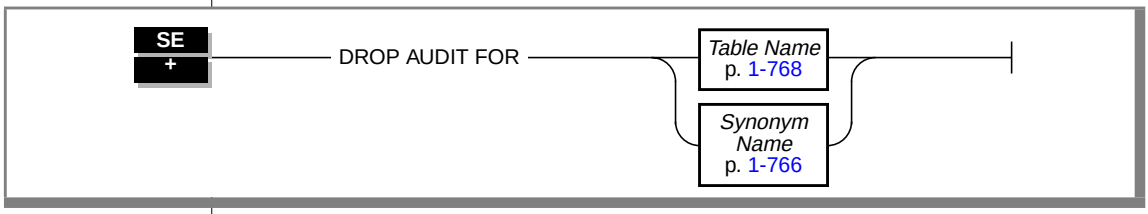
See the CONNECT, SET CONNECTION, and DATABASE statements in this manual.

For information on multithreaded applications, see the *INFORMIX-ESQL/C Programmer's Manual*.

DROP AUDIT

Use the DROP AUDIT statement to delete an audit-trail file.

Syntax



Usage

When you finish making a backup of your database files, use the DROP AUDIT statement to remove the old audit-trail file. Use the CREATE AUDIT statement to start a new audit trail for a table.

You must own the table or have the DBA privilege to use the DROP AUDIT statement.

The following example assumes that you have just backed up the **stores7** database. It removes the existing audit trail for the **orders** table.

```
DROP AUDIT FOR orders
```

References

See the CREATE AUDIT and RECOVER TABLE statements in this manual.

In the *INFORMIX-SE Administrator's Guide*, see the discussion on audit trails.

DROP DATABASE

Use the DROP DATABASE statement to delete an entire database, including all system catalog tables, indexes, and data.

Syntax

Usage

You must have the DBA privilege or be user **informix** to run the DROP DATABASE statement successfully. Otherwise, the database server issues an error message and does not drop the database.

You cannot drop the current database or a database that is being used by another user. All the database users must first execute the CLOSE DATABASE statement.

The (The)Tj9 0 0 9 192.87 345.33 Tm-0.037e5 9 s*q174 6302f-4.78893ease nabamulti174 630cute the

You can specify the full pathname of the database in quotes, as the following example shows:

```
DROP DATABASE '/u/training/stores7'
```

You cannot use a ROLLBACK WORK statement to undo a DROP DATABASE statement. If you roll back a transaction that contains a DROP DATABASE statement, the database is not re-created, and you do not receive an error message. ♦

SE
ESQL

You can specify a database that is not in your local directory or DBPATH by putting the full operating-system file in a variable for the database name, as the following example shows:

```
LET db_var = '/u/training/stores7'
DROP DATABASE db_var
```

♦

DB

Use this statement with caution. DB-Access does not prompt you to verify that you want to delete the entire database. ♦

ESQL

You can use a simple database name in a program or host variable, or you can use the full database server and database name. See “Database Name” on page 1-660 for more information. ♦

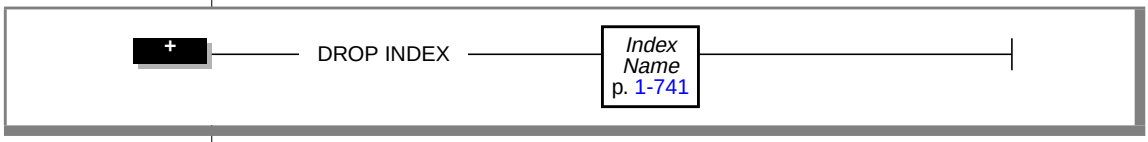
References

See the CREATE DATABASE and CLOSE DATABASE statements in this manual.

DROP INDEX

Use the DROP INDEX statement to remove an index.

Syntax



Usage

You must be the owner of the index or have the DBA privilege to use the DROP INDEX statement.

The following example drops the index **o_num_ix** that **joed** owns. The **stores7** database must be the current database.

```
DROP INDEX stores7:joed.o_num_ix
```

You cannot use the DROP INDEX statement on a column or columns to drop a unique constraint that is created with a CREATE TABLE statement; you must use the ALTER TABLE statement to remove indexes that are created as constraints with a CREATE TABLE or ALTER TABLE statement.

The index is not actually dropped if it is shared by constraints. Instead, it is renamed in the **sysindexes** system catalog table with the following format:

```
[space]<tabid>_<constraint id>
```

In this example, *tabid* and *constraint_id* are from the **systables** and **sysconstraints** system catalog tables, respectively. The **idxname** (index name) column in the **sysconstraints** table is then updated to reflect this change. For example, the renamed index name might be something like this: “121_13” (quotes used to show the spaces).

If this index is a unique index with only referential constraints sharing it, the index is downgraded to a duplicate index after it is renamed.

SE

You cannot use a ROLLBACK WORK statement to undo a DROP INDEX statement. If you roll back a transaction that contains a DROP INDEX statement, the index is not re-created, and you do not receive an error message. ♦

References

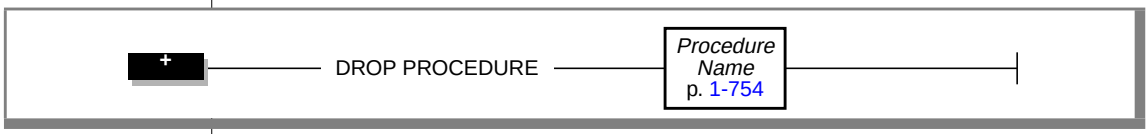
See the ALTER TABLE, CREATE INDEX, and CREATE TABLE statements in this manual.

In the *INFORMIX-OnLine Dynamic Server Performance Guide*, see the discussion of indexes.

DROP PROCEDURE

Use the DROP PROCEDURE statement to remove a stored procedure from the database.

Syntax



Usage

You must be the owner of the stored procedure or have the DBA privilege to use the DROP PROCEDURE statement.

Dropping the stored procedure removes the text and executable versions of the procedure.

SE

You cannot use a ROLLBACK WORK statement to undo a DROP PROCEDURE statement. If you roll back a transaction that contains a DROP PROCEDURE statement, the stored procedure is not re-created, and you do not receive an error message. ♦

Tip: *You cannot drop a stored procedure from within the same procedure.*

References

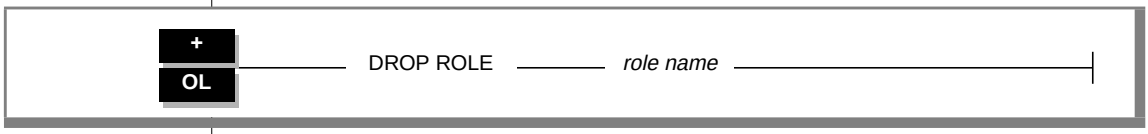
See the CREATE PROCEDURE statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of using stored procedures in Chapter 12.

DROP ROLE

Use the DROP ROLE statement to remove a previously created role.

Syntax



Element	Purpose	Restrictions	Syntax
role name	Name of the role being dropped	The role name must have been created with the CREATE ROLE statement	Identifier, p. 1-723

Usage

The DROP ROLE statement is used to remove an existing role. Either the DBA or a user to whom the role was granted with the WITH GRANT OPTION can issue the DROP ROLE statement.

After a role is dropped, the privileges associated with that role, such as table-level privileges or fragment-level privileges, are dropped, and a user cannot grant or enable a role. If a user is using the privileges of a role when the role is dropped, the user automatically loses those privileges.

A role exists until either the DBA or a user to whom the role was granted with the WITH GRANT OPTION uses the DROP ROLE statement to drop the role.

The following statement drops the role **engineer**:

```
DROP ROLE engineer
```

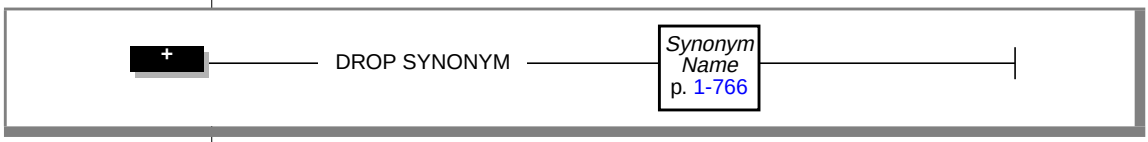
References

See the CREATE ROLE, GRANT, REVOKE, and SET ROLE statements in this manual.

DROP SYNONYM

Use the DROP SYNONYM statement to remove a previously defined synonym.

Syntax



Usage

You must be the owner of the synonym or have the DBA privilege to use the DROP SYNONYM statement.

The following statement drops the synonym **nj_cust**, which **cathyg** owns:

```
DROP SYNONYM cathyg.nj_cust
```

If a table is dropped, any synonyms that are in the same database as the table and that refer to the table are also dropped.

If a synonym refers to an external table, and the table is dropped, the synonym remains in place until you explicitly drop it using DROP SYNONYM. You can create another table or synonym in place of the dropped table and give the new object the name of the dropped table. The old synonym then refers to the new object. See the CREATE SYNONYM statement for a complete discussion of synonym chaining.

SE

You cannot use a ROLLBACK WORK statement to undo a DROP SYNONYM statement. If you roll back a transaction that contains a DROP SYNONYM statement, the synonym is not re-created, and you do not receive an error message. ♦

References

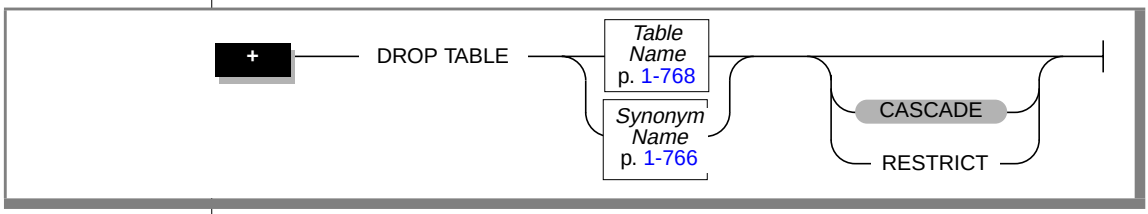
See the CREATE SYNONYM statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of synonyms in Chapter 11.

DROP TABLE

Use the DROP TABLE statement to remove a table, along with its associated indexes and data.

Syntax



Usage

You must be the owner of the table or have the DBA privilege to use the DROP TABLE statement.

DB

If you issue a DROP TABLE statement, you are not prompted to verify that you want to delete an entire table. ♦

SE

You cannot use a ROLLBACK WORK statement to undo a DROP TABLE statement. If you roll back a transaction that contains a DROP TABLE statement, the table is not re-created, and you do not receive an error message. ♦

Effects of DROP TABLE Statement

Use the DROP TABLE statement with caution. When you remove a table, you also delete the data stored in it, the indexes or constraints on the columns (including all the referential constraints placed on its columns), any local synonyms assigned to it, any triggers created for it, and any authorizations you have granted on the table. You also drop all views based on the table and any violations and diagnostics tables associated with the table. You do not remove any synonyms for the table that have been created in an external database.

Specifying CASCADE Mode

The CASCADE mode means that a DROP TABLE statement removes related database objects, including referential constraints built on the table, views defined on the table, and any violations and diagnostics tables associated with the table. The CASCADE mode is the default mode of the DROP TABLE statement. You can also specify this mode explicitly with the CASCADE keyword.

Specifying RESTRICT Mode

With the RESTRICT keyword, you can control the success or failure of the drop operation for tables that have referential constraints and views defined on the table or have violations and diagnostics tables associated with the table. Using the RESTRICT option causes the drop operation to fail and an error message to be returned if any existing referential constraints reference *table name* or if any existing views are defined on *table name* or if any violations and diagnostics tables are associated with *table name*.

Tables That Cannot Be Dropped

Observe the following restrictions on the types of tables that you can drop:

- You cannot drop any system catalog tables.
- You cannot drop a table that is not in the current database.
- You cannot drop a violations or diagnostics table. Before you can drop such a table, you must first issnBef Et393.4Et30 387.99 60ST.2 T33 VIOLA

Examples of Dropping a Table

The following example deletes two tables. Both tables are within the current database and are owned by the current user. Neither table has a violations or diagnostics table associated with it. Neither table has a referential constraint or view defined on it.

```
DROP TABLE customer;
DROP TABLE stores7@accntg:joed.state;
```

References

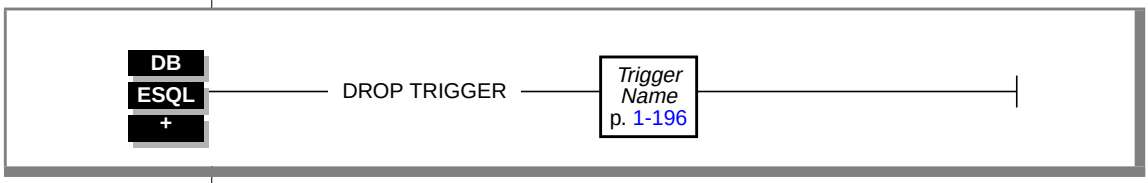
See the CREATE TABLE and DROP DATABASE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of data integrity and creating a table in Chapter 4 and Chapter 9, respectively.

DROP TRIGGER

Use the DROP TRIGGER statement to remove a trigger definition from the database.

Syntax



Usage

You must be the owner of the trigger or have the DBA privilege to drop a trigger.

Dropping a trigger removes the text of the trigger definition and the executable trigger from the database.

The following statement drops the **items_pct** trigger:

```
DROP TRIGGER items_pct
```

You cannot drop a trigger inside a stored procedure if the procedure is called within a data manipulation statement. For example, in the following INSERT statement, a DROP TRIGGER statement is illegal inside the stored procedure **proc1**:

```
INSERT INTO orders EXECUTE PROCEDURE proc1(vala, valb)
```

SE

You cannot use a ROLLBACK WORK statement to undo a DROP TRIGGER statement. If you roll back a transaction that contains a DROP TRIGGER statement, the trigger is not re-created, and you do not receive an error message. ♦

References

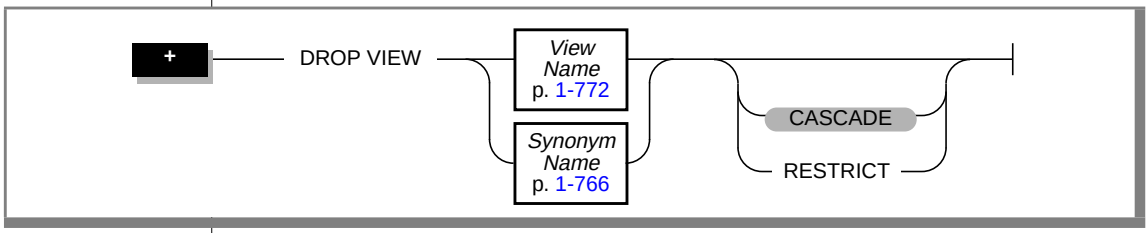
See the CREATE PROCEDURE statement in this manual for more information about a stored procedure that is called within a data manipulation statement.

For more information about triggers, see the CREATE TRIGGER statement in this manual.

DROP VIEW

Use the DROP VIEW statement to remove a view from the database.

Syntax



Usage

You must own the view or have the DBA privilege to use the DROP VIEW statement.

When you drop *view name*, you also drop all views that have been defined in terms of that view. You can also specify this default condition with the CASCADE keyword.

When you use the RESTRICT keyword in the DROP VIEW statement, the drop operation fails if any existing views are defined on *view name*, which would be abandoned in the drop operation.

You can query the **sysdepend** system catalog table to determine which views, if any, depend on another view.

The following statement drops the view that is named **cust1**:

```
DROP VIEW cust1
```

SE

You cannot use a ROLLBACK WORK statement to undo a DROP VIEW statement. If you roll back a transaction that contains a DROP VIEW statement, the view is not re-created, and you do not receive an error message. ♦

References

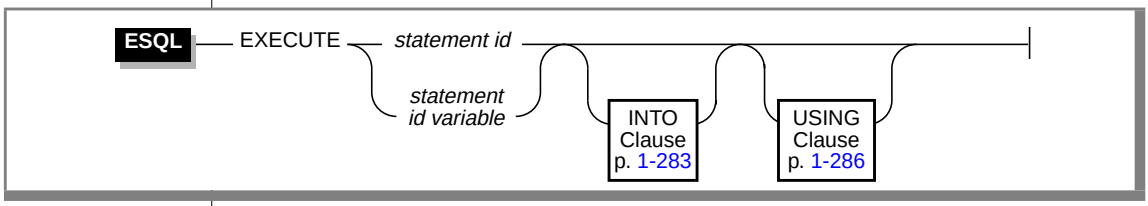
See the CREATE VIEW and DROP TABLE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of views in Chapter 10.

EXECUTE

Use the EXECUTE statement to run a previously prepared statement or set of statements.

Syntax



Element	Purpose	Restrictions	Syntax
statement id	Identifies an SQL statement	You must have defined the statement identifier in a previous PREPARE statement. After you release the database server resources (using a FREE statement), you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.	PREPARE, p. 1-402
statement id variable	Host variable that identifies an SQL statement	You must have defined the host variable in a previous PREPARE statement. The host variable must be a character data type.	PREPARE, p. 1-402

Usage

The EXECUTE statement passes a prepared SQL statement to the database server for execution. If the statement contained question mark (?) placeholders, specific values are supplied for them before execution. Once prepared, an SQL statement can be executed as often as needed.

You can execute any prepared statement. However, for stored procedures that return more than one row, you cannot execute a prepared SELECT statement or a prepared EXECUTE PROCEDURE statement. When you use a prepared SELECT statement to return multiple rows of data, you can use the DECLARE, OPEN, and FETCH cursor statements to retrieve the data rows. In addition, you can use EXECUTE on a prepared SELECT INTO TEMP statement to achieve the same result. If you prepare an EXECUTE PROCEDURE statement for a procedure that returns multiple rows, you need to use the DECLARE, OPEN, and FETCH cursor statements just as you would with a SELECT statement.

If you create or drop a trigger after you prepared a triggering INSERT, DELETE, or UPDATE statement, the prepared statement returns an error when you execute it.

The following example shows an EXECUTE statement within an INFORMIX-ESQL/C program:

```
EXEC SQL prepare del_1 from
    'delete from customer
       where customer_num = 119';
EXEC SQL execute del_1;
```

Scope of Statement Identifiers

A program can consist of one or more source-code files. By default, the scope of a statement identifier is global to the program, so a statement identifier created in one file can be referenced from another file.

In a multiple-file program, if you want to limit the scope of a statement identifier to the file in which it is executed, you can preprocess all the files with the **-local** command-line option. See your SQL API product manual for more information, restrictions, and performance issues when you preprocess files with the **-local** option.

The sqlca Record and EXECUTE

Following an EXECUTE statement, the **sqlca** (see your SQL API product manual) can reflect two results:

- The **sqlca** can reflect an error within the EXECUTE statement. For example, when an UPDATE ... WHERE ... statement within a prepared object processes zero rows, the database server sets **sqlca** to 100.
- The **sqlca** can also reflect the success or failure of the executed statement.

INTO Clause

Element	Purpose	Restrictions	Syntax
output descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-757
output descriptor variable	Host variable name that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-757
output indicator variable	Host variable that receives a return code if null data is placed in the corresponding *output variable*	Variable cannot be DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.
output sqlda pointer	Points to an **sqlda** structure that defines the data type and memory location of values that correspond to the question-mark (?) placeholder in a prepared statement.	You cannot begin an output sqlda pointer with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	DESCRIBE, p. 1-255
output variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

The INTO clause allows you to execute a prepared singleton SELECT statement or a prepared EXECUTE PROCEDURE statement, and store the returned values into output variables, output SQL descriptors, or output **sqlda** pointers. The INTO clause provides a concise and efficient alternative to more complicated and lengthy syntax. In addition, by placing values into variables that can be displayed, the INTO clause simplifies and enhances your ability to retrieve and display data values. For example, if you use the INTO clause, you do not have to use the PREPARE, DECLARE, OPEN, and FETCH sequence of statements to retrieve values from a table.

Important: *If you execute a prepared SELECT statement that returns more than one row of data, you receive an error message. In addition, if you prepare and declare a statement, and then attempt to execute that statement, you receive an error message.*

You cannot select a null value from a table column and place that value into an output variable. If you know in advance that a table column contains a null value, make sure after you select the data that you check the indicator variable that is associated with the column to determine if the value is null.

The following list describes the procedure for using the INTO clause with the EXECUTE statement:

1. Declare the output variables that the EXECUTE statement uses.
2. Use the PREPARE statement to prepare your SELECT statement or to prepare your EXECUTE PROCEDURE statement.
3. Use the EXECUTE statement, with the INTO clause, to execute your SELECT statement or to execute your EXECUTE PROCEDURE statement.

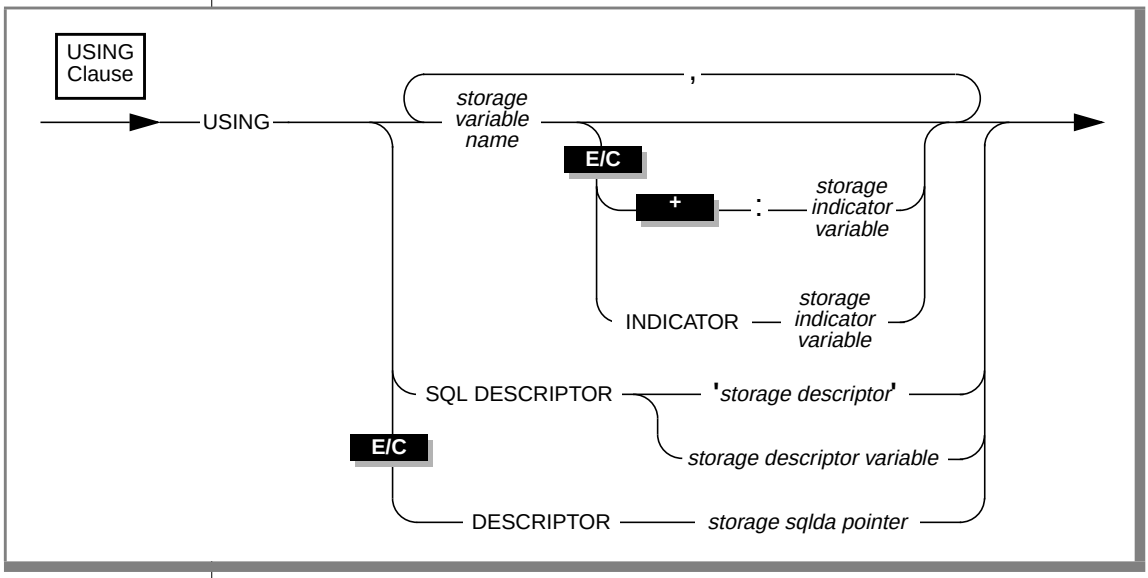
The following example shows how to use the INTO clause with an EXECUTE statement in INFORMIX-ESQL/C:

```
EXEC SQL prepare sel1 from 'select fname, lname from customer
    where customer_num =123';
EXEC SQL execute sel1 into :fname, :lname using :cust_num;
```

The following example shows how to use the INTO clause to return multiple rows of data:

```
EXEC SQL BEGIN DECLARE SECTION;
int customer_num =100;
char fname[25];
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare sel1 from 'select fname from customer
    where customer_num=?';
for ( ;customer_num < 200; customer_num++)
    {
    EXEC SQL execute sel1 into :fname using customer_num;
    printf("Customer number is %d\n", customer_num);
    printf("Customer first name is %s\n\n", fname);
    }
```

USING Clause



Element	Purpose	Restrictions	Syntax
storage descriptor	Quoted string that identifies a system-descriptor area	System-descriptor area must already be allocated. Make sure surrounding quotes are single.	Quoted String, p. 1-757
storage descriptor variable	Host variable name that identifies a system-descriptor area	System-descriptor area must already be allocated.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
storage indicator variable	Host variable that receives a return code if null data is placed in the corresponding *data variable*. Receives truncation information if truncation occurs.	Variable cannot be DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.
storage sqlda pointer	Points to an **sqlda** structure that defines the data type and memory location of values that correspond to the question-mark (?) placeholder in a prepared statement.	You cannot begin *storage sqlda pointer* with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	DESCRIBE, p. 1-255
storage variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

The USING clause specifies values that are to replace question-mark (?) placeholders in the prepared statement. Providing values in the EXECUTE statement that replace the question-mark placeholders in the prepared statement is sometimes called *parameterizing* the prepared statement.

You can specify any of the following items to replace the question-mark placeholders in a statement before you execute it:

- A host variable name (if the number and data type of the question marks are known at compile time)
- A system descriptor that identifies a system
- A descriptor that is a pointer to an **sqlda** structure ♦

E/C

Supplying Parameters Through Host or Program Variables

You must supply one storage variable name for each placeholder. The data type of each variable must be compatible with the corresponding value that the prepared statement requires.

The following example executes the prepared UPDATE statement in INFORMIX-ESQL / C:

```
stcopy ("update orders set order_date = ? where po_num = ?", stm1);
EXEC SQL prepare statement_1 from :stm1;
EXEC SQL execute statement_1 using :order_date :po_num;
```

Supplying Parameters Through a System Descriptor

You can create a system-descriptor area that describes the data type and memory location of one or more values and then specify the storage descriptor in the USING SQL DESCRIPTOR clause of the EXECUTE statement.

Each time that the EXECUTE statement is run, the values that the system-descriptor area describes are used to replace question-mark (?) placeholders in the PREPARE statement.

The COUNT field corresponds to the number of dynamic parameters in the prepared statement. The value of COUNT must be less than or equal to the value of the occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

For more information on system descriptors, see your SQL API product manual.

The following examples show how to use system descriptors to execute prepared statements in INFORMIX-ESQL / C and INFORMIX-ESQL / COBOL, respectively:

INFORMIX-ESQL/C

```
EXEC SQL execute prep_stmt using sql descriptor 'desc1';
```

INFORMIX-ESQL/COBOL

```
EXEC SQL EXECUTE PREP_STMT USING SQL DESCRIPTOR 'DESC1'
    END-EXEC.
```

Supplying INFORMIX-ESQL/C Parameters Through an sqlda Structure

You can specify the storage **sqlda** pointer in the USING DESCRIPTOR clause of the EXECUTE statement. Each time the EXECUTE statement is run, the values that the **sqlda** structure describes are used to replace question-mark (?) placeholders in the PREPARE statement.

For more information on the **sqlda** structure, see the manual for the version of INFORMIX-ESQL/C that you are using.

The following example shows how to use an **sqlda** structure to execute a prepared statement in INFORMIX-ESQL/C:

```
EXEC SQL execute prep_stmt using descriptor pointer2
```

Error Conditions with EXECUTE

In a database that is not ANSI compliant, if any statement fails to access any rows, the database server returns `(0)`.

ANSI

In an ANSI-compliant database, if you prepare and execute any of the following statements, and no rows are returned, the database server returns `SQLNOTFOUND (100)`:

- INSERT INTO *table-name* SELECT ... WHERE ...
- SELECT INTO TEMP ... WHERE ...
- DELETE ... WHERE
- UPDATE ... WHERE ... ♦

In a multistatement prepare, if any statement in the preceding list fails to access rows, in either ANSI databases or databases that are not ANSI compliant, the database server returns `SQLNOTFOUND (100)`.

References

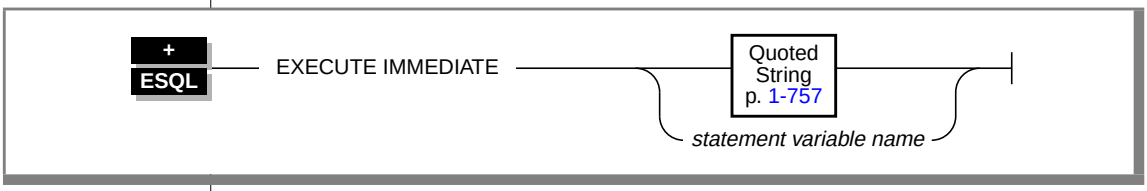
See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, EXECUTE IMMEDIATE, GET DESCRIPTOR, PREPARE, PUT, and SET DESCRIPTOR statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the EXECUTE statement in Chapter 5.

EXECUTE IMMEDIATE

Use the EXECUTE IMMEDIATE statement to perform the functions of the PREPARE, EXECUTE, and FREE statements.

Syntax



Element	Purpose	Restrictions	Syntax
statement variable name	Host variable whose value is a character string that consists of one or more SQL statements.	The host variable must have been defined within the program. The variable must be character data type. For additional restrictions, see "EXECUTE IMMEDIATE and Restricted Statements" on page 1-291 and "Restrictions on Allowed Statements" on page 1-291.	Variable name must conform to language-specific rules for variable names.

Usage

The quoted string is a character string that includes one or more SQL statements. The string, or the contents of *statement variable name*, is parsed and executed if correct; then all data structures and memory resources are released immediately. In the usual method of dynamic execution, these functions are distributed among the PREPARE, EXECUTE, and FREE statements.

The EXECUTE IMMEDIATE statement makes it easy to execute dynamically a single simple SQL statement, which is constructed during program execution. For example, you could obtain the name of a database from program input, construct the DATABASE statement as a program variable, and then use EXECUTE IMMEDIATE to execute the statement, which opens the database.

EXECUTE IMMEDIATE and Restricted Statements

You cannot use the EXECUTE IMMEDIATE statement to execute the following SQL statements.

CLOSE	GET DESCRIPTOR
CONNECT	OPEN
DECLARE	PREPARE
DISCONNECT	SELECT
EXECUTE	SET CONNECTION
EXECUTE PROCEDURE (if the procedure returns values)	SET DESCRIPTOR
FETCH	WHENEVER
GET DIAGNOSTICS	

Use a PREPARE statement to execute a dynamically constructed SELECT statement.

Restrictions on Allowed Statements

The following restrictions apply to the statement that is contained in the quoted string or in *statement variable name*:

- The statement cannot contain a host-language comment.
- Names of host-language variables are not recognized as such in prepared text. The only identifiers that you can use are names defined in the database, such as table names and columns.
- The statement cannot reference a host variable list or a descriptor; it must not contain any question-mark (?) placeholders, which are allowed with a PREPARE statement.
- The text must not include any embedded SQL statement prefix or terminator, such as the dollar sign ($), colon (:), or the words EXEC SQL.

Example of the EXECUTE IMMEDIATE Statement

The following example shows the EXECUTE IMMEDIATE statement in INFORMIX-ESQL / C:

```
sprintf(cdb_text, "create database %s", usr_db_id);
EXEC SQL execute immediate :cdb_text;
```

References

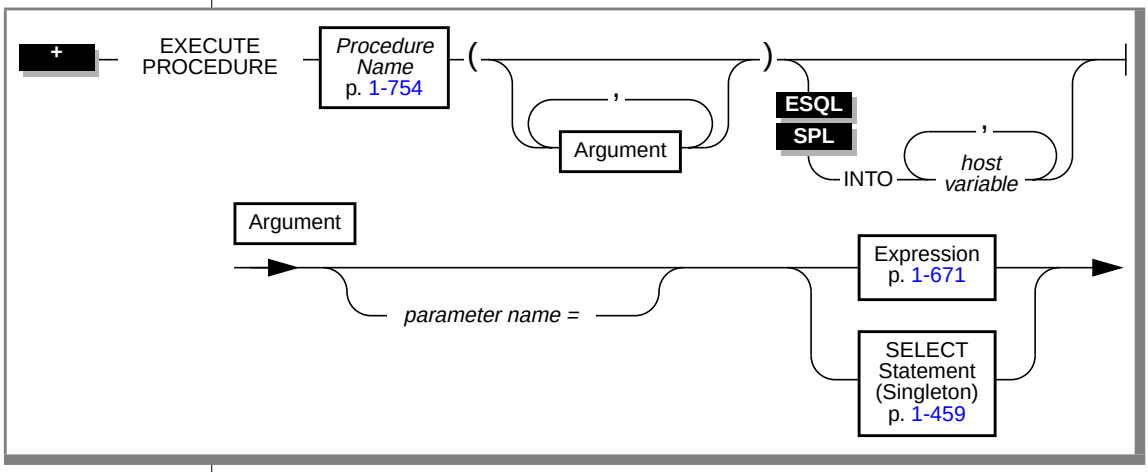
See the EXECUTE, FREE, and PREPARE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of quick execution in Chapter 5.

EXECUTE PROCEDURE

Use the EXECUTE PROCEDURE statement to execute a procedure from the DB-Access interactive editor, an SQL API, or another stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
host variable	A host variable that receives a returned value from a procedure, or a list of such variables	If you issue an EXECUTE PROCEDURE statement within an SQL API, the receiving variables must be host variables. If you issue an EXECUTE PROCEDURE statement within a stored procedure, the receiving variables must be procedure variables. If you issue an EXECUTE PROCEDURE statement within a CREATE TRIGGER statement, the receiving variables must be column names within the triggering table or another table.	The name of a host variable must conform to language-specific rules for variable names. For the syntax of procedure variables, see Expression, p. 1-671. For the syntax of column names, see Identifier, p. 1-723.
parameter name	The name of a parameter for which you supply an argument to the procedure	The parameter name must match the parameter name that you specified in a corresponding CREATE PROCEDURE statement. If you use the *parameter name* = syntax for any argument in the EXECUTE PROCEDURE statement, you must use it for all arguments.	Expression, p. 1-671

Usage

The EXECUTE PROCEDURE statement invokes a procedure called *Procedure Name*.

If an EXECUTE PROCEDURE statement has more arguments than the called procedure expects, an error is returned.

If an EXECUTE PROCEDURE statement has fewer arguments than the called procedure expects, the arguments are said to be missing. Missing arguments are initialized to their corresponding default values if default values were specified. (See the CREATE PROCEDURE statement on page 1-134.) This initialization occurs before the first executable statement in the body of the procedure.

If arguments are missing and do not have default values, they are initialized to the value of UNDEFINED. An attempt to use any variable that has the value of UNDEFINED results in an error.

Name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use *parameter name* = syntax for none or all of the arguments that are specified in one EXECUTE PROCEDURE statement.

For instance, in the following example, both the procedure calls are valid for a procedure that expects three character arguments, `t`, `d`, and `n`:

```
EXECUTE PROCEDURE add_col (t ='customer', d ='integer', n ='newint')
EXECUTE PROCEDURE add_col ('customer','newint','integer')
```

ESQL

If the EXECUTE PROCEDURE statement returns more than one row, it must be enclosed within an SPL FOREACH loop or accessed through a cursor. ♦

INTO Clause

ESQL

SPL

The INTO clause specifies where the values that the procedure returns will be stored.

The *host variable* list is a list of the host variables that receive the returned values from a procedure call. A procedure that returns more than one row must be enclosed in a cursor.

If you execute a procedure from within a stored procedure, the list contains procedure variables.

If you execute a procedure from within a triggered action, the list contains column names from the triggering table or another table. For information on triggered actions, see the CREATE TRIGGER statement on page 1-192.

You cannot prepare an EXECUTE PROCEDURE statement that has an INTO clause. See “Executing Stored Procedures Within a PREPARE Statement” on page 1-406 for more information. ♦

References

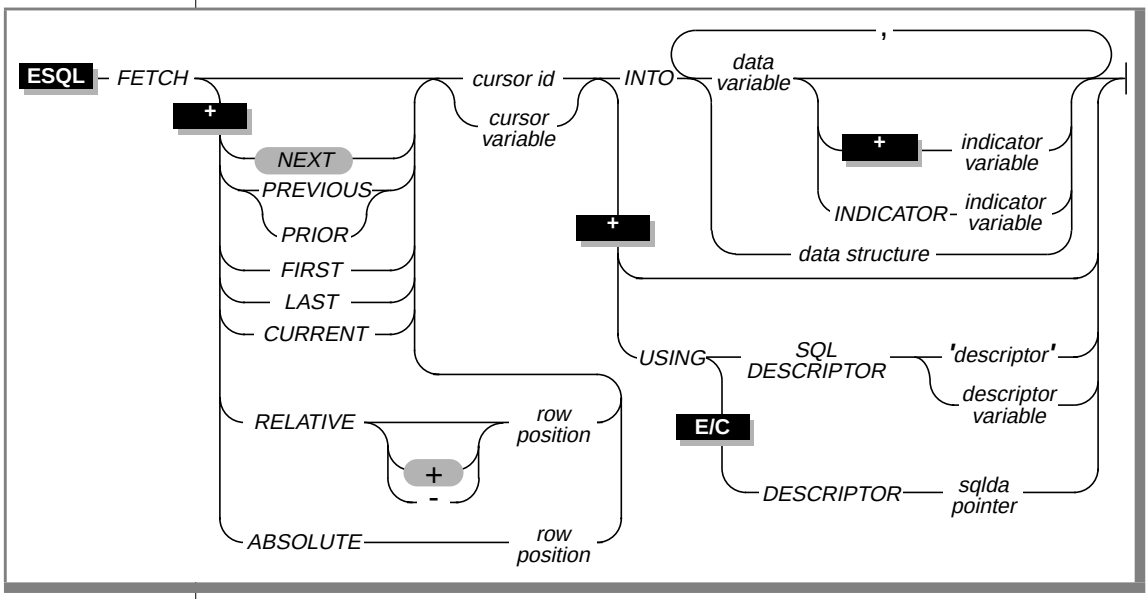
See the CREATE PROCEDURE, DROP PROCEDURE, GRANT, and CALL statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of creating and using stored procedures in Chapter 12.

FETCH

Use the FETCH statement to move a cursor to a new row in the active set and to retrieve the row values from memory.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	Identifier for a cursor from which rows are to be retrieved	The cursor must have been created in an earlier DECLARE statement and opened in an earlier OPEN statement.	Identifier, p. 1-723
cursor variable	Host variable name that holds the value of *cursor id*	The cursor identified in *cursor variable* must have been created in an earlier DECLARE statement and opened in an earlier OPEN statement.	Variable name must conform to language-specific rules for variable names.
data structure	Structure that has been declared as a host variable	The individual members of the data structure must be matched appropriately to the type of values that are being fetched. If you use a program array, you must list both the array name and a specific element of the array in *data structure*.	Data-structure name must conform to language-specific rules for data-structure names.
data variable	Host variable that receives one value from the fetched row	The host variable must have a data type that is appropriate for the value that is fetched into it.	Variable name must conform to language-specific rules for variable names.
descriptor	String that identifies the system-descriptor area into which you fetch the contents of a row	The system-descriptor area must have been allocated with the ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-757
descriptor variable	Host variable name that holds the value of *descriptor*	The system-descriptor area that is identified in *descriptor variable* must have been allocated with the ALLOCATE DESCRIPTOR statement.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
indicator variable	Host variable that receives a return code if null data is placed in the corresponding *data variable*	This parameter is optional, but use an indicator variable if the possibility exists that the value of *data variable* is null. If you specify the indicator variable without the INDICATOR keyword, you cannot put a space between *data variable* and *indicator variable*. The rules for placing a prefix before *indicator variable* are language-specific. See your SQL API manual for further information on indicator variables.	Variable name must conform to language-specific rules for variable names.
row position	Integer value or host variable that contains an integer value. The integer value gives the position of the desired row in the active set of rows. See "FETCH with a Scroll Cursor" on page 1-300 for a discussion of the RELATIVE and ABSOLUTE keywords and the meaning of *row position* with each keyword.	A value of 0 for *row position* is allowed with the RELATIVE keyword. A value of 0 fetches the current row. The value of *row position* must be 1 or higher with the ABSOLUTE keyword.	If you are using a host variable, variable name must conform to language-specific rules for variable names. If you are using a literal number, see Literal Number, p. 1-752.
sqlda pointer	Pointer to an **sqlda** structure that receives the values from the fetched row	You cannot begin an **sqlda** pointer with a dollar sign ($) or a colon (:).	See the discussion of **sqlda** structure in the *INFORMIX-ESQL/C Programmer's Manual*.

(2 of 2)

Usage

The FETCH statement is one of four statements that are used for queries that return more than one row from the database. The four statements, DECLARE, OPEN, FETCH, and CLOSE, are used in the following sequence:

1. Declare a cursor to control the active set of rows.
2. Open the cursor to begin execution of the query.
3. Fetch from the cursor to retrieve the contents of each row.

4. Close the cursor to break the association between the cursor and the active set.

A cursor is created as either a sequential cursor or a scroll cursor. The way the database server creates and stores members of the active set and then fetches rows from the active set differs depending on whether the cursor is a sequential cursor or a scroll cursor. (See the DECLARE statement on page 1-234 for details on the types of cursors.)

X/O

In X/Open mode, if a cursor-direction value (such as NEXT or RELATIVE) is specified, a warning message is issued, indicating that the statement does not conform to X/Open standards. ♦

FETCH with a Sequential Cursor

A sequential cursor can fetch only the next row in sequence from the active set. The sole keyword option that is available to a sequential cursor is the default value, NEXT. A sequential cursor can read through a table only once each time it is opened. The following example in INFORMIX-ESQL/C illustrates the use of a sequential cursor:

```
EXEC SQL fetch seq_curs into :fname, :lname;
```

When the program opens a sequential cursor, the database server processes the query to the point of locating or constructing the first row of data. The goal of the database server is to tie up as few resources as possible.

Because the sequential cursor can retrieve only the next row, the database server can frequently create the active set one row at a time. On each FETCH operation, the database server returns the contents of the current row and locates the next row. This one-row-at-a-time strategy is not possible if the database server must create the entire active set to determine which row is the first row (as would be the case if the SELECT statement included an ORDER BY clause).

FETCH with a Scroll Cursor

A scroll cursor can fetch any row in the active set, either by specifying an absolute row position or a relative offset. Use the following keywords to specify a particular row that you want to retrieve.

Keyword	Effect
NEXT	retrieves the next row in the active set.
PREVIOUS	retrieves the previous row in the active set.
PRIOR	is synonymous with PREVIOUS; it retrieves the previous row in the active set.
FIRST	retrieves the first row in the active set.
LAST	retrieves the last row in the active set.
CURRENT	retrieves the current row in the active set (the same row as returned by the preceding FETCH statement from the scroll cursor).
RELATIVE	retrieves the *n*th row, relative to the current cursor position in the active set, where *row position* supplies *n*. A negative value indicates the *n*th row prior to the current cursor position. If *row position* is 0, the current row is fetched.
ABSOLUTE	retrieves the *n*th row in the active set, where *row position* supplies *n*. Absolute row positions are numbered from *1*.

The following INFORMIX-ESQL/C examples illustrate the FETCH statement:

```
EXEC SQL fetch previous q_curs into :orders;

EXEC SQL fetch last q_curs into :orders;

EXEC SQL fetch relative -10 q_curs into :orders;

printf("Which row? ");
scanf("%d",row_num);
EXEC SQL fetch absolute :row_num q_curs into :orders;
```

Row Numbers

The row numbers that are used with the ABSOLUTE keyword are valid only while the cursor is open. Do not confuse them with rowid values. A rowid value is based on the position of a row in its table and remains valid until the table is rebuilt. A row number for a FETCH statement is based on the position of the row in the active set of the cursor; the next time the cursor is opened, different rows might be selected.

How the Database Server Stores Rows

The database server must retain all the rows in the active set for a scroll cursor until the cursor closes, because it cannot be sure which row the program asks for next. When a scroll cursor opens, the database server implements the active set as a temporary table although it might not fill this table immediately.

The first time a row is fetched, the database server copies it into the temporary table as well as returning it to the program. When a row is fetched for the second time, it can be taken from the temporary table. This scheme uses the fewest resources in case the program abandons the query before it fetches all the rows. Rows that are never fetched are usually not created or are saved in a temporary table.

Specifying Where Values Go in Memory

Each value from the select list of the query or the output of the executed procedure must be returned into a memory location. You can specify these destinations in one of the following ways:

- Use the INTO clause of a SELECT statement.
- Use the INTO clause of a EXECUTE PROCEDURE statement.
- Use the INTO clause of a FETCH statement.
- Use a system-descriptor area.
- Use an **sqlda** structure. ♦

E/C

Using the INTO Clause of SELECT

The SELECT statement that is associated with the cursor can contain an INTO clause, which specifies the program variables that are to receive the values. You can use this method only when the SELECT statement is written as part of the declaration of the cursor (see the DECLARE statement on page 1-234). In this case, the FETCH statement cannot contain an INTO clause. The following example uses the INTO clause of the SELECT statement to specify program variables in INFORMIX-ESQL / C:

```
EXEC SQL declare ord_date cursor for
    select order_num, order_date, po_num
        into :o_num, :o_date, :o_po;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

Use an indicator variable if the data that is returned from the SELECT statement might be null. See your SQL API manual for more information about indicator variables.

Using the INTO Clause of EXECUTE PROCEDURE

The EXECUTE PROCEDURE statement that is associated with the cursor can contain an INTO clause, which specifies the program variables that are to receive the values. You can use this method only when the EXECUTE PROCEDURE statement is written as part of the cursor declaration (see the DECLARE statement on page 1-234). In this case, the FETCH statement cannot contain an INTO clause. The following example uses the INTO clause of the EXECUTE PROCEDURE statement to specify program variables in INFORMIX-ESQL / C:

```
EXEC SQL declare ord_date cursor for
    execute procedure xx (20)
        into :o_num, :o_date, :o_po;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

Use an indicator variable if the data that is returned from the EXECUTE PROCEDURE statement might be null. See your SQL API manual for more information about indicator variables.

Using the INTO Clause of FETCH

When the SELECT statement omits the INTO clause, you must specify the destination of the data whenever a row is fetched. The FETCH statement can include an INTO clause to retrieve data into a set of variables. This method lets you store different rows in different memory locations.

In the following INFORMIX-ESQL/C example, a series of complete rows is fetched into a program array. The INTO clause of each FETCH statement specifies an array element as well as the array name.

```
EXEC SQL BEGIN DECLARE SECTION;
    char wanted_state[2];
    short int row_count = 0;
    struct customer_t{
    {
        int    c_no;
        char   fname[15];
        char   lname[15];
    } cust_rec[100];
EXEC SQL END DECLARE SECTION;

main()
{
    EXEC SQL connect to'stores7';
    printf("Enter 2-letter state code: ");
    scanf ("%s", wanted_state);

    EXEC SQL declare cust cursor for
        select * from customer where state = :wanted_state;

    EXEC SQL open cust;

    EXEC SQL fetch cust into :cust_rec[row_count];
    while (SQLCODE == 0)
    {
        printf("\n%s %s", cust_rec[row_count].fname,
            cust_rec[row_count].lname);

        row_count++;
        EXEC SQL fetch cust into :cust_rec[row_count];
    }
    printf ("\n");
    EXEC SQL close cust;
    EXEC SQL free cust;
}
```

You can fetch into a program-array element only by using an INTO clause in the FETCH statement. When you are declaring a cursor, do not refer to an array element within the SQL statement.

Using a System-Descriptor Area

You can use a system-descriptor area as an output variable. The keywords USING SQL DESCRIPTOR introduce the name of the system-descriptor area into which you fetch the contents of a row. You can then use the GET DESCRIPTOR statement to transfer the values that the FETCH statement returns from the system-descriptor area into host variables.

For more information, see the manual for your SQL API. The following examples show sample FETCH USING SQL DESCRIPTOR statements in INFORMIX-ESQL / C and INFORMIX-ESQL / COBOL, respectively:

INFORMIX-ESQL/C

```
EXEC SQL fetch selcurs using sql descriptor 'desc';
```

INFORMIX-ESQL/COBOL

```
EXEC SQL FETCH SEL_CURS USING SQL DESCRIPTOR 'DESC' END-EXEC.
```

Using an sqlda Structure

E/C

You can use a pointer to an **sqlda** structure to supply destinations. This structure contains data descriptors that specify the data type and memory location for one selected value. For more information, see the *INFORMIX-ESQL/C Programmer's Manual*. The keywords USING DESCRIPTOR introduce the name of the **sqlda** pointer structure.

When you create a SELECT statement dynamically, you cannot use an INTO *host-variable* clause because you cannot name host variables in a prepared statement. If you are certain of the number and data type of values in the select list, you can use an INTO *host-variable* clause in the FETCH statement. However, if user input generated the query, you might not be certain of the number and data type of values that are being selected. In this case, you must use an **sqlda** pointer structure, as the following list describes:

- Use the DESCRIBE statement to fill in the **sqlda** structure.
- Allocate memory to hold the data values.
- Name the **sqlda** structure in the FETCH statement.

The following example shows a sample FETCH USING DESCRIPTOR statement in INFORMIX-ESQL / C:

```
EXEC SQL fetch selcurs using descriptor pointer2;
```

♦

Fetching a Row for Update

The FETCH statement does not ordinarily lock a row that is fetched. Thus, another process can modify (update or delete) the fetched row immediately after your program receives it. A fetched row is locked in the following cases:

- When you set the isolation level to Repeatable Read, each row you fetch is locked with a read lock to keep it from changing until the cursor closes or the current transaction ends. Other programs can also read the locked rows.
- When you set the isolation level to Cursor Stability, the current row is locked.
- ANSI In an ANSI-compliant database, an isolation level of Repeatable Read is the default; you can set it to something else. ♦
- When you are fetching through an update cursor (one that is declared FOR UPDATE), each row you fetch is locked with a promotable lock. Other programs can read the locked row, but no other program can place a promotable or write lock; therefore, the row is unchanged if another user tries to modify it using the WHERE CURRENT OF clause of UPDATE or DELETE statement.

When you modify a row, the lock is upgraded to a write lock and remains until the cursor is closed or the transaction ends. If you do not modify it, the lock might or might not be released when you fetch another row, depending on the isolation level you have set. The lock on an unchanged row is released as soon as another row is fetched, unless you are using Repeatable Read isolation (see the SET ISOLATION statement on page 1-575).

Important: *You can hold locks on additional rows even when Repeatable Read isolation is not in use or is unavailable. Update the row with unchanged data to hold it locked while your program is reading other rows. You must evaluate the effect of this technique on performance in the context of your application, and you must be aware of the increased potential for deadlock.*

When you use explicit transactions, be sure that a row is both fetched and modified within a single transaction; that is, both the FETCH statement and the subsequent UPDATE or DELETE statement must fall between a BEGIN WORK statement and the next COMMIT WORK statement.

SE

You cannot set the database isolation level for INFORMIX-SE. ♦

Checking the Result of FETCH

You can use the GET DIAGNOSTICS statement to check the result of each FETCH statement. Examine the **RETURNED_SQLSTATE** field of the GET DIAGNOSTICS statement to check if the field contains the value `02000`.

If a row is returned successfully, the **RETURNED_SQLSTATE** field of GET DIAGNOSTICS contains the value `00000`. If no row is found, the preprocessor sets the **SQLSTATE** code to `02000`, which indicates `no data found`, and the current row is unchanged. Five conditions set the **SQLSTATE** code to `02000`, indicating `no data found`, as the following list describes:

- The active set contains no rows.
- You issue a FETCH NEXT statement when the cursor points to the last row in the active set or points past it.
- You issue a FETCH PRIOR or FETCH PREVIOUS statement when the cursor points to the first row in the active set.
- You issue a FETCH RELATIVE *n* statement when no *n*th row exists in the active set.
- You issue a FETCH ABSOLUTE *n* statement when no *n*th row exists in the active set.

See the GET DIAGNOSTICS statement in this manual for more information.

You can also use SQLCODE of **sqlca** to determine the same results. See the *Informix Guide to SQL: Tutorial* for further information about SQLCODE of **sqlca**.

References

See the ALLOCATE DESCRIPTOR, CLOSE, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, GET DESCRIPTOR, OPEN, PREPARE, and SET DESCRIPTOR statements in this manual for further information about using the FETCH statement with dynamic management statements.

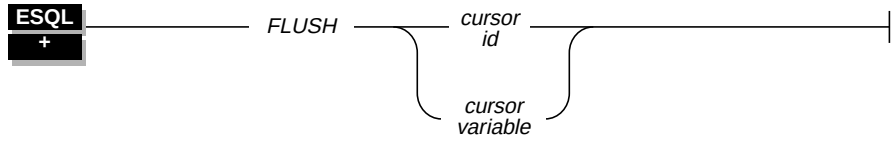
In the *Informix Guide to SQL: Tutorial*, see the discussion of the FETCH statement in Chapter 5.

For further information about error checking and the system-descriptor area, see your SQL API manual.

FLUSH

Use the FLUSH statement to force rows that a PUT statement buffered to be written to the database.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	Identifies a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 1-723
cursor variable	Host variable that identifies a cursor	Host variable must be a character data type. A DECLARE statement must have previously created the cursor.	Variable name must conform to language-specific rules for variable names.

Usage

The PUT statement adds a row to a buffer, and the buffer is written to the database when it is full. Use the FLUSH statement to force the insertion when the buffer is not full.

If the program terminates without closing the cursor, the buffer is left unflushed. Rows placed into the buffer since the last flush are lost. Do not expect the end of the program to close the cursor and flush the buffer.

The following example shows a FLUSH statement:

```
FLUSH icurs
```

Error Checking FLUSH Statements

The **sqlca** structure contains information on the success of each FLUSH statement and the number of rows that are inserted successfully. The result of each FLUSH statement is contained in the fields of the **sqlca**, as the following table shows.

ESQL/C	ESQL/COBOL
sqlca.sqlcode, SQLCODE	SQLCODE of SQLCA
sqlca.sqlerrd[2]	SQLERRD[3] OF SQLCA

When you use data buffering with an insert cursor, you do not discover errors until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is flushed. When an error is discovered, rows in the buffer that are located after the error are *not* inserted; they are lost from memory.

The **SQLCODE** field is set either to an error code or to zero if no error occurs. The third element of the **sqlerrd** array is set to the number of rows that are successfully inserted into the database:

- If a block of rows is successfully inserted into the database, **SQLCODE** is set to zero and **sqlerrd** to the count of rows.
- If an error occurs while the FLUSH statement is inserting a block of rows, **SQLCODE** shows which error, and **sqlerrd** contains the number of rows that were successfully inserted. (Uninserted rows are discarded from the buffer.)

Tip: *When you encounter an **SQLCODE** error, a corresponding **SQLSTATE** error value might exist. Check the GET DIAGNOSTICS statement for information about getting the **SQLSTATE** value and using the GET DIAGNOSTICS statement to interpret the **SQLSTATE** value.*

Counting Total and Pending Rows

To count the number of rows actually inserted into the database as well as the number not yet inserted, perform the following steps:

1. Prepare two integer variables, for example, **total** and **pending**.
2. When the cursor opens, set both variables to 0.
3. Each time a PUT statement executes, increment both **total** and **pending**.
4. Whenever a FLUSH statement executes or the cursor is closed, subtract the third field of the SQLERRD array from **pending**.

References

See the CLOSE, DECLARE, OPEN, and PUT statements in this manual.

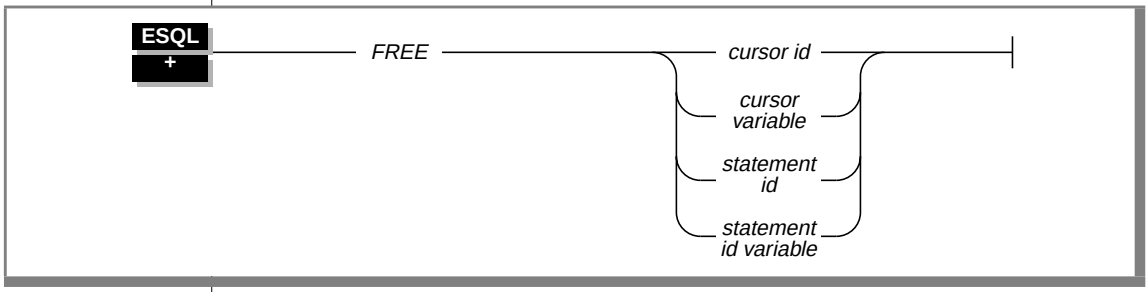
For information about the **sqlca** structure, see your SQL API manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of FLUSH in Chapter 6.

FREE

The FREE statement releases resources that are allocated to a prepared statement or to a cursor.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	Identifies a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 1-723
cursor variable	Host variable that identifies a cursor	Variable must be a character data type. Cursor must have been previously created by a DECLARE statement.	Variable name must conform to language-specific rules for variable names
statement id	Identifies an SQL statement	The statement identifier must be defined in a previous PREPARE statement. After you release the database-server resources, you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.	PREPARE, p. 1-402
statement id variable	A host variable that identifies an SQL statement	This variable must be defined in a previous PREPARE statement. Variable must be a character data type.	PREPARE, p. 1-402

Usage

The FREE statement releases the resources that were allocated for a prepared statement or a declared cursor in the application-development tool and the database server. Resources are allocated when you prepare a statement or when you open a cursor (see the DECLARE and OPEN statements on pages 1-234 and 1-390, respectively.)

The amount of available memory in the system limits the total number of open cursors and prepared statements that are allowed at one time in one process. Use FREE *statement id* or FREE *statement id variable* to release the resources that a prepared statement holds; use FREE *cursor id* or FREE *cursor variable* to release resources that a cursor holds.

Freeing a Statement

If you prepared a statement (but did not declare a cursor for it), FREE *statement id* (or *statement id variable*) releases the resources in both the application development tool and the database server.

If you declared a cursor for a prepared statement, FREE *statement id* (or *statement id variable*) releases only the resources in the application development tool; the cursor can still be used. The resources in the database server are released only when you free the cursor.

After you free a statement, you cannot execute it or declare a cursor for it until you prepare it again.

The following INFORMIX-ESQL / C example shows the sequence of statements that is used to free an implicitly prepared statement:

```
EXEC SQL prepare sel_stmt from 'select * from orders';
.
.
.
EXEC SQL free sel_stmt;
```

The following INFORMIX-ESQL / C example shows the sequence of statements that are used to release the resources of an explicitly prepared statement. The first FREE statement in this example frees the cursor. The second FREE statement in this example frees the prepared statement.

```
sprintf(demoselect, "%s %s",
    "select * from customer ",
    "where customer_num between 100 and 200");
EXEC SQL prepare sel_stmt from :demoselect;
EXEC SQL declare sel_curs cursor for sel_stmt;
EXEC SQL open sel_curs;
 .
 .
 .
EXEC SQL close sel_curs;
EXEC SQL free sel_curs;
EXEC SQL free sel_stmt;
```

Freeing a Cursor

If you declared a cursor for a prepared statement, freeing the cursor releases only the resources in the database server. To release the resources for the statement in the application-development tool, use FREE *statement id* (or *statement id variable*).

If a cursor is not declared for a prepared statement, freeing the cursor releases the resources in both the application-development tool and the database server.

After a cursor is freed, it cannot be opened until it is declared again. The cursor should be explicitly closed before it is freed.

For an example of a FREE statement that frees a cursor, see the second example in "Freeing a Statement" on page 1-312.

References

See the CLOSE, DECLARE, EXECUTE, EXECUTE IMMEDIATE, and PREPARE statements in this manual.

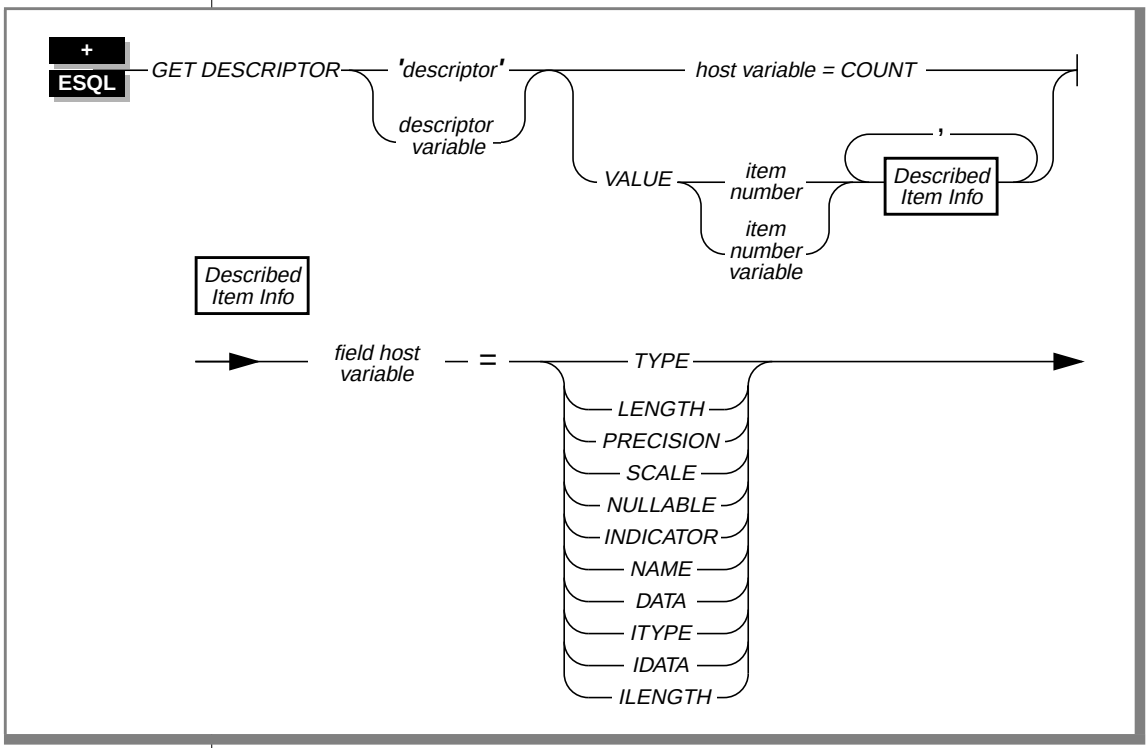
In the *Informix Guide to SQL: Tutorial*, see the discussion of the FREE statement in Chapter 5.

GET DESCRIPTOR

Use the GET DESCRIPTOR statement to accomplish the following separate tasks:

- Determine how many values are described in a system-descriptor area by retrieving the value in the COUNT field.
- Determine the characteristics of each column or expression that is described in the system-descriptor area.
- Copy a value from the system-descriptor area into a host variable after a FETCH statement.

Syntax



Element	Purpose	Restrictions	Syntax
descriptor	A quoted string that identifies a system-descriptor area from which information is to be obtained	The system-descriptor area must have been allocated in an ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-757
descriptor variable	An embedded variable name that holds the value of *descriptor*	The system-descriptor area identified in *descriptor variable* must have been allocated in an ALLOCATE DESCRIPTOR statement.	The name of the embedded variable must conform to language-specific rules for variable names.
field host variable	The name of a host variable that receives the contents of the specified field from the system-descriptor area	The *field host variable* must be an appropriate type to receive the value of the specified field from the system-descriptor area	The name of the *field host variable* must conform to language-specific rules for variable names.
host variable	The name of a *host variable* that indicates how many values are described in the system-descriptor area	The host variable must be an integer data type.	The name of the *host variable* must conform to language-specific rules for variable names.
item number	An unsigned integer that represents one of the occurrences (item descriptors) in the system-descriptor area	The value of *item number* must be greater than zero and less than the number of occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 1-752
item number variable	The name of a host variable that holds the value of *item number*	The *item number variable* must be an integer data type.	The name of the *item number variable* must conform to language-specific rules for variable names.

Usage

If an error occurs during the assignment to any identified host variable, the contents of the host variable are undefined.

The role and contents of each field in a system-descriptor area are described in the SQL API manuals.

The GET DESCRIPTOR statement can be used in EXECUTE PROCEDURE statements, which have been described with the USING SQL DESCRIPTOR parameter.

The host variables that are used in the GET DESCRIPTOR statement must be declared in the BEGIN DECLARE SECTION of a SQL API program. See your SQL API manual for specifics.

Using the COUNT Keyword

Use the COUNT keyword to determine how many values are described in the system-descriptor area.

The following INFORMIX-ESQL/C example shows how to use a GET DESCRIPTOR statement with a host variable to determine how many values are described in the system-descriptor area called **desc1**:

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
int h_count;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc1' with max occurrences 20;

/* This section of program would prepare a SELECT or INSERT
 * statement into the s_id statement id.
*/
EXEC SQL describe s_id using sql descriptor 'desc1';

EXEC SQL get descriptor 'desc1' :h_count = count;
...
}
```

VALUE Clause

Use the VALUE clause to obtain information about a described column or expression or to retrieve values that the database server returns.

The *item number* must be greater than zero and less than the number of occurrences that were specified when the system-descriptor area was allocated using ALLOCATE DESCRIPTOR.

Using the VALUE Clause After a DESCRIBE

After you describe a SELECT, EXECUTE PROCEDURE, or INSERT statement, the characteristics of each column or expression in the select list of the SELECT statement, the characteristics of the values returned by the EXECUTE PROCEDURE statement, or the characteristics of each column in the INSERT statement are returned to the system-descriptor area. Each value in the system-descriptor area describes the characteristics of one returned column or expression. Each field and its possible contents are described in your SQL API manuals.

The following INFORMIX-ESQL/C example shows how to use a GET DESCRIPTOR statement to obtain data type information from the **demodesc** system-descriptor area:

```
EXEC SQL get descriptor 'demodesc' value :index
        :type = TYPE,
        :len = LENGTH,
        :name = NAME;
printf("Column %d: type = %d, len = %d, name = %s\n",
        index, type, len, name);
```

The value that the database server returns into the TYPE field is a defined integer. To evaluate the data type that is returned, test for a specific integer value. The codes for the TYPE field are listed in the manual for your SQL API. For additional information about integer data type values, see page 1-544.

X/O

In X/Open mode, the X/Open code is returned to the TYPE field. You cannot mix the two modes because errors can result. For example, if a particular data type is not defined under X/Open mode but is defined for Informix products, executing a GET DESCRIPTOR statement can result in an error.

In X/Open mode, a warning message appears if ILENGTH, IDATA, or ITYPE is used. It indicates that these fields are not standard X/Open fields for a system-descriptor area.

For more information about TYPE, ILENGTH, IDATA, and ITYPE, see the related dynamic management chapter in the appropriate Informix SQL API programmer's manual. For more information about programming in X/Open mode, see the preprocessing and compilation syntax in the appropriate Informix SQL API programmer's manual. ♦

If the TYPE of a fetched value is DECIMAL or MONEY, the database server returns the precision and scale information for a column into the **PRECISION** and **SCALE** fields after a DESCRIBE statement is executed. If the TYPE is *not* DECIMAL or MONEY, the **SCALE** and **PRECISION** fields are undefined.

Using the VALUE Clause After a FETCH

Each time your program fetches a row, it must copy the fetched value into host variables so that the data can be used. To accomplish this task, use a GET DESCRIPTOR statement after each fetch of each value in the select list. If three values exist in the select list, you need to use three GET DESCRIPTOR statements after each fetch (assuming you want to read all three values). The item numbers for each of the three GET DESCRIPTOR statements are 1, 2, and 3.

The following INFORMIX-ESQL/C example shows how you can copy data from the **DATA** field into a host variable (**result**) after a fetch. For this example, it is predetermined that all returned values are the same data type.

```
EXEC SQL get descriptor 'demodesc' :desc_count = count;
.
.
.
EXEC SQL fetch democursor using sql descriptor 'demodesc';
for (i = 1; i <= desc_count; i++)
    {
    if (sqlca.sqlcode != 0) break;
    EXEC SQL get descriptor 'demodesc' value :i :result = DATA;
    printf("%s ", result);
    }
printf("\n");
```

The following INFORMIX-ESQL/COBOL example shows how you can copy data from the DATA field into host variables after a fetch. The first GET DESCRIPTOR statement uses a literal item number; the second GET DESCRIPTOR statement uses a host variable to hold the item number.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 COUNT    SQLINT.
    01 ITEMNO   SQLINT.
    01 TYPE     SQLINT.
    01 LENGTH   SQLINT.
    01 LONGVAL  SQLINT.
    01 CHVAL    SQLCHAR(21).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL GET DESCRIPTOR 'desc1' VALUE 1
     :TYPE = TYPE, :LENGTH = LENGTH, :CHVAL = DATA
END-EXEC.

MOVE 2 TO ITEMNO.
EXEC SQL GET DESCRIPTOR 'desc1' VALUE :ITEMNO.
         :TYPE = TYPE, :LONGVAL = DATA
         END-EXEC.
```

Fetching a Null Value

When you use GET DESCRIPTOR after a fetch, and the fetched value is null, the **INDICATOR** field is set to -1 (NULL). The value of DATA is undefined if **INDICATOR** indicates a null value. The host variable into which DATA is copied has an unpredictable value.

Using LENGTH or ILENGTH

If your **DATA** or **IDATA** field contains a character string, you must specify a value for **LENGTH**. If you specify LENGTH=0, **LENGTH** is automatically set to the maximum length of the string. The **DATA** or **IDATA** field might contain a literal character string or a character string that is derived from a character variable of CHAR or VARCHAR data type. This provides a method to determine dynamically the length of a string in the **DATA** or **IDATA** field.

If a DESCRIBE statement precedes a GET DESCRIPTOR statement, **LENGTH** is automatically set to the maximum length of the character field that is specified in your table.

This information is identical for **ILENGTH**. Use **ILENGTH** when you create a dynamic program that does not comply with the X/Open standard.

References

See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, OPEN, PREPARE, PUT, and SET DESCRIPTOR statements in this manual for more information about using dynamic SQL statements.

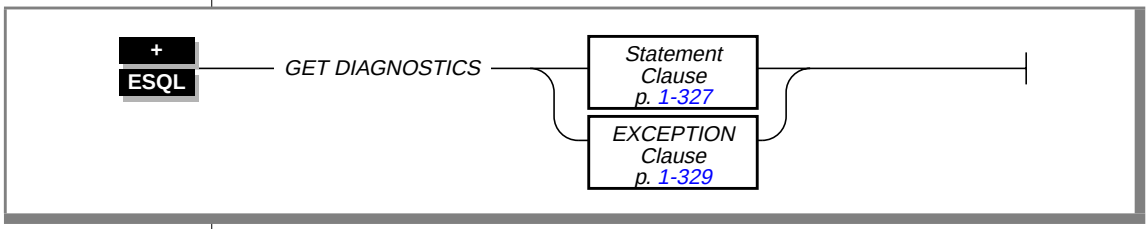
For more information about the system-descriptor area, see your SQL API manual.

GET DIAGNOSTICS

Use the GET DIAGNOSTICS statement to return diagnostic information about executing an SQL statement. The GET DIAGNOSTICS statement uses one of two clauses, as described in the following list:

- The Statement clause determines count and overflow information about errors and warnings generated by the most recent SQL statement.
- The EXCEPTION clause provides specific information about errors and warnings generated by the most recent SQL statement.

Syntax



Usage

The GET DIAGNOSTICS statement retrieves selected status information from the diagnostics area and retrieves either count and overflow information or information on a specific exception.

The GET DIAGNOSTICS statement never changes the contents of the diagnostics area.

Using the SQLSTATE Error Status Code

When an SQL statement executes, an error status code is automatically generated. This code represents `success`, `failure`, `warning`, or `no data found`. This error status code is stored in a variable called **SQLSTATE**.

Class and Subclass Codes

The **SQLSTATE** status code is a a five-character string that can contain only digits and capital letters.

The first two characters of the **SQLSTATE** status code indicate a class. The last three characters of the **SQLSTATE** code indicate a subclass. Figure 1-3 shows the structure of the **SQLSTATE** code. This example uses the value `08001`, where `08` is the class code and `001` is the subclass code. The value `08001` represents the error `unable to connect with database environment`.

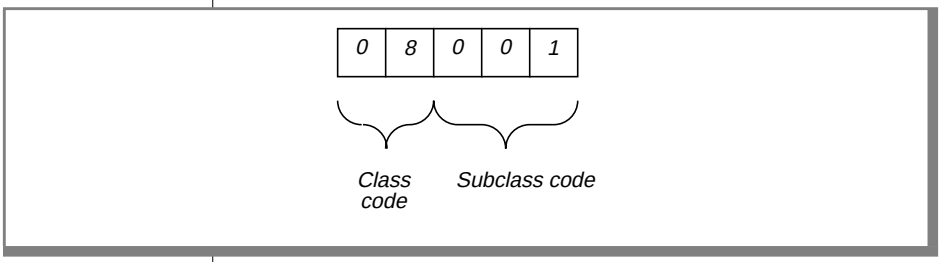


Figure 1-3
The Structure of the SQLSTATE Code

The following table is a quick reference for interpreting class code values.

SQLSTATE Class Code Value	Outcome
00	Success
01	Success with warning
02	No data found
> 02	Error or warning

Support for ANSI Standards

All status codes returned to the **SQLSTATE** variable are ANSI compliant except in the following cases:

- **SQLSTATE** codes with a class code of `01` and a subclass code that begins with a `I` are Informix-specific warning messages.
- **SQLSTATE** codes with a class code of `IX` and any subclass code are Informix-specific error messages.

- **SQLSTATE** codes whose class code begins with a digit in the range 5 to 9 or with a capital letter in the range I to Z indicate conditions that are currently undefined by ANSI. The only exception is that **SQLSTATE** codes whose class code is IX are Informix-specific error messages.

List of SQLSTATE Codes

The following table describes the class codes, subclass codes, and the meaning of all valid warning and error codes associated with the **SQLSTATE** error status code.

Class	Subclass	Meaning
00	000	Success
01	000	Success with warning
01	002	Disconnect error. Transaction rolled back
01	003	Null value eliminated in set function
01	004	String data, right truncation
01	005	Insufficient item descriptor areas
01	006	Privilege not revoked
01	007	Privilege not granted
01	I01	Database has transactions
01	I03	ANSI-compliant database selected
01	I04	INFORMIX-OnLine database selected
01	I05	Float to decimal conversion has been used
01	I06	Informix extension to ANSI-compliant standard syntax
01	I07	UPDATE/DELETE statement does not have a WHERE clause
01	I08	An ANSI keyword has been used as a cursor name
01	I09	Number of items in the select list is not equal to the number in the into list
01	I10	Database server running in secondary mode
01	I11	Dataskip is turned on
02	000	No data found

(1 of 4)

Class	Subclass	Meaning
07	000	Dynamic SQL error
07	001	USING clause does not match dynamic parameters
07	002	USING clause does not match target specifications
07	003	Cursor specification cannot be executed
07	004	USING clause is required for dynamic parameters
07	005	Prepared statement is not a cursor specification
07	006	Restricted data type attribute violation
07	008	Invalid descriptor count
07	009	Invalid descriptor index
08	000	Connection exception
08	001	Server rejected the connection
08	002	Connection name in use
08	003	Connection does not exist
08	004	Client unable to establish connection
08	006	Transaction rolled back
08	007	Transaction state unknown
08	S01	Communication failure
0A	000	Feature not supported
0A	001	Multiple server transactions
21	000	Cardinality violation
21	S01	Insert value list does not match column list
21	S02	Degree of derived table does not match column list

(2 of 4)

Class	Subclass	Meaning
22	000	Data exception
22	001	String data, right truncation
22	002	Null value, no indicator parameter
22	003	Numeric value out of range
22	005	Error in assignment
22	027	Data exception trim error
22	012	Division by zero
22	019	Invalid escape character
22	024	Unterminated string
22	025	Invalid escape sequence
23	000	Integrity constraint violation
24	000	Invalid cursor state
25	000	Invalid transaction state
2B	000	Dependent privilege descriptors still exist
2D	000	Invalid transaction termination
26	000	Invalid SQL statement identifier
2E	000	Invalid connection name
28	000	Invalid user-authorization specification
33	000	Invalid SQL descriptor name
34	000	Invalid cursor name
35	000	Invalid exception number
37	000	Syntax error or access violation in PREPARE or EXECUTE IMMEDIATE
3C	000	Duplicate cursor name
40	000	Transaction rollback
40	003	Statement completion unknown

(3 of 4)

Class	Subclass	Meaning
42	000	Syntax error or access violation
S0	000	Invalid name
S0	001	Base table or view table already exists
S0	002	Base table not found
S0	011	Index already exists
S0	021	Column already exists
S1	001	Memory allocation failure
IX	000	Informix reserved error message

(4 of 4)

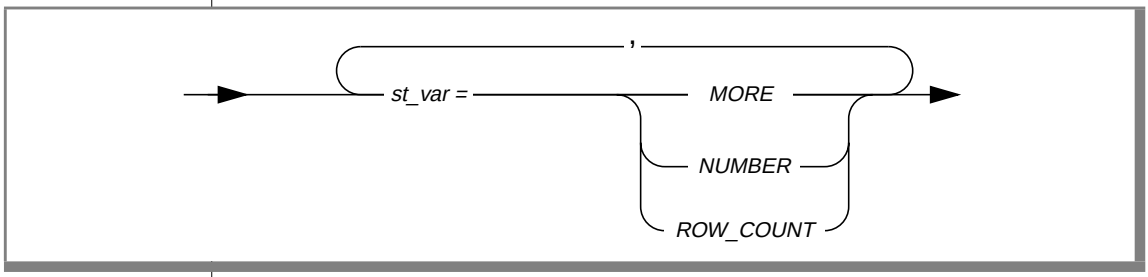
Using SQLSTATE in Applications

You can use a variable, called **SQLSTATE**, that you do not have to declare in your program. **SQLSTATE** contains the error code that is essential for error handling, which is generated every time your program executes an SQL statement. **SQLSTATE** is created automatically. You can examine the **SQLSTATE** variable to determine whether an SQL statement was successful. If the **SQLSTATE** variable indicates that the statement failed, you can execute a GET DIAGNOSTICS statement to obtain additional error information.

For an example of how to use an **SQLSTATE** variable in a program, see "Using GET DIAGNOSTICS for Error Checking" on page 1-336.

In Chapter 5 of the *Informix Guide to SQL: Tutorial*, see the discussion about error-code handling. In addition, refer to the error-handling chapter of your SQL API manual.

The Statement Clause

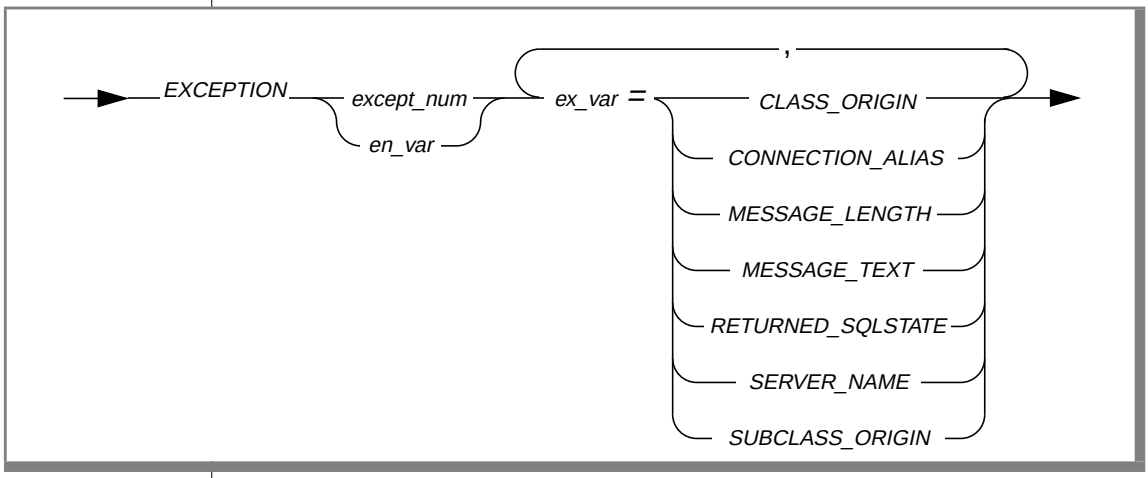


Element	Purpose	Restrictions	Syntax
st_var	Host variable that receives status information about the most recent SQL statement. Receives information for the specified status field name.	Data type must match that of the requested field.	Variable name must conform to language-specific rules for variable names.

When retrieving count and overflow information, GET DIAGNOSTICS can deposit the values of the three statement fields into corresponding host variable. The host-variable data type must be the same as that of the requested field. These three fields are represented by the following keywords.

Field Name Keyword	Field Data Type	Field Contents	ESQL/C Host Variable Data Type	ESQL/COBOL Host Variable Data Type
MORE	Character	Y or N	char[2]	PIC X(1)
NUMBER	Integer	1 to 35,000	int	PIC S9(9)
ROW_COUNT	Integer	0 to 999,999,999	int	PIC S9(9)

The EXCEPTION Clause



Element	Purpose	Restrictions	Syntax
en_var	Host variable that specifies an exception number for a GET DIAGNOSTICS statement	Variable must contain an integer value limited to a range from 1 to 35,000. Variable data type must be INT or SMALLINT.	Variable name must conform to language-specific rules for variable names.
except_num	Literal integer value that specifies the exception number for a GET DIAGNOSTICS statement. The *except_num* literal indicates one of the exception values from the number of exceptions returned by the NUMBER field in the Statement clause.	Integer value is limited to a range from 1 to 35,000.	Literal Number, p. 1-752
ex_var	Host variable that you declare, which receives EXCEPTION information about the most recent SQL statement. Receives information for a specified exception field name.	Data type must match that of the requested field.	Variable name must conform to language-specific rules for variable names.

When retrieving exception information, GET DIAGNOSTICS deposits the values of each of the seven fields into corresponding host variables. These fields are located in the diagnostics area and are derived from an exception raised by the most recent SQL statement.

The host-variable data type must be the same as that of the requested field. The seven exception information fields are represented by the keywords described in the following table.

Field Name Keyword	Field Data Type	Field Contents	ESQL/C Host Variable Data Type	ESQL/COBOL Host Variable Data Type
RETURNED_SQLSTATE	Character	SQLSTATE value	char[6]	PIC X(5)
CLASS_ORIGIN	Character	String	char[255]	PIC X(254)
SUBCLASS_ORIGIN	Character	String	char[255]	PIC X(254)
MESSAGE_TEXT	Character	String	char[255]	PIC X(254)
MESSAGE_LENGTH	Integer	Numeric value	int	PIC 9(4) COMP-5
SERVER_NAME	Character	String	char[255]	PIC X(254)
CONNECTION_NAME	Character	String	char[255]	PIC X(254)

The application specifies the exception by number, using either an unsigned integer or an integer host variable (an exact numeric with a scale of 0). An exception with a value of 1 corresponds to the **SQLSTATE** value set by the most recent SQL statement other than GET DIAGNOSTICS. The association between other exception numbers and other exceptions raised by that SQL statement is undefined. Thus, no set order exists in which the diagnostic area can be filled with exception values. You always get at least one exception, even if the **SQLSTATE** value indicates success.

If an error occurs within the GET DIAGNOSTICS statement (that is, if an illegal exception number is requested), the Informix internal **SQLCODE** and **SQLSTATE** variables are set to the value of that exception. In addition, the GET DIAGNOSTICS fields are undefined.

Using the RETURNED_SQLSTATE Keyword

Use the RETURNED_SQLSTATE keyword to determine the **SQLSTATE** value that describes the exception.

Using the CLASS_ORIGIN Keyword

Use the CLASS_ORIGIN keyword to retrieve the class portion of the **RETURNED_SQLSTATE** value. If the International Standards Organization (ISO) standard defines the class, the value of CLASS_ORIGIN is equal to ISO 9075. Otherwise, the value of CLASS_ORIGIN is defined by Informix and cannot be ISO 9075. ANSI SQL and ISO SQL are synonymous.

Using the SUBCLASS_ORIGIN Keyword

Use the SUBCLASS_ORIGIN keyword to define the source of the subclass portion of the **RETURNED_SQLSTATE** value. If the ISO international standard defines the subclass, the value of SUBCLASS_ORIGIN is equal to ISO 9075.

Using the MESSAGE_TEXT Keyword

Use the MESSAGE_TEXT keyword to determine the message text of the exception (for example, an error message).

Using the MESSAGE_LENGTH Keyword

Use the MESSAGE_LENGTH keyword to determine the length of the current MESSAGE_TEXT string.

Using the SERVER_NAME Keyword

Use the SERVER_NAME keyword to determine the name of the database server associated with the actions of a CONNECT or DATABASE statement.

When the SERVER_NAME Field Is Updated

The GET DIAGNOSTICS statement updates the **SERVER_NAME** field when the following situations occur:

- A CONNECT statement successfully executes.
- A SET CONNECTION statement successfully executes.
- A DISCONNECT statement successfully executes at the current connection.
- A DISCONNECT ALL statement fails.

When the SERVER_NAME Field Is Not Updated

The **SERVER_NAME** field is not updated when:

- a CONNECT statement fails.
- a DISCONNECT statement fails (this does not include the DISCONNECT ALL statement).
- a SET CONNECTION statement fails.

The **SERVER_NAME** field retains the value set in the previous SQL statement. If any of the preceding conditions occur on the first SQL statement that executes, the **SERVER_NAME** field is blank.

The Contents of the SERVER_NAME Field

The **SERVER_NAME** field contains different information after you execute the following statements.

Executed Statement	SERVER_NAME Field Contents
CONNECT	It contains the name of the database server to which you connect or fail to connect. Field is blank if you do not have a current connection or if you make a default connection.
SET CONNECTION	It contains the name of the database server to which you switch or fail to switch.
DISCONNECT	It contains the name of the database server from which you disconnect or fail to disconnect. If you disconnect and then you execute a DISCONNECT statement for a connection that is not current, the **SERVER_NAME** field remains unchanged.
DISCONNECT ALL	It sets the field to blank if the statement executes successfully. If the statement does not execute successfully, the **SERVER_NAME** field contains the names of all the database servers from which you did not disconnect. However, this information does not mean that the connection still exists.

If the CONNECT statement is successful, the **SERVER_NAME** field is set to one of the following values:

- The **INFORMIXSERVER** value if the connection is to a default database server (that is, the CONNECT statement does not list a database server).
- The name of the specific database server if the connection is to a specific database server.

The DATABASE Statement

When you execute a DATABASE statement, the **SERVER_NAME** field contains the name of the server on which the database resides.

Using the CONNECTION_NAME Keyword

Use the CONNECTION_NAME keyword to specify a name for the connection used in your CONNECT or DATABASE statements.

When the CONNECTION_NAME Keyword Is Updated

GET DIAGNOSTICS updates the **CONNECTION_NAME** field when the following situations occur:

- A CONNECT statement successfully executes.
- A SET CONNECTION statement successfully executes.
- A DISCONNECT statement successfully executes at the current connection. GET DIAGNOSTICS fills the **CONNECTION_NAME** field with blanks because no current connection exists.
- A DISCONNECT ALL statement fails.

When CONNECTION_NAME Is Not Updated

The **CONNECTION_NAME** field is not updated when the following situations occur:

- A CONNECT statement fails.
- A DISCONNECT statement fails (this does not include the DISCONNECT ALL statement).
- A SET CONNECTION statement fails.

The **CONNECTION_NAME** field retains the value set in the previous SQL statement. If any of the preceding conditions occur on the first SQL statement that executes, the **CONNECTION_NAME** field is blank.

The Contents of the CONNECTION_NAME Field

The **CONNECTION_NAME** field contains different information after you execute the following statements.

Executed Statement	CONNECTION_NAME Field Contents
CONNECT	It contains the name of the connection, specified in the CONNECT statement, to which you connect or fail to connect. The field is blank if you do not have a current connection or if you make a default connection.
SET CONNECTION	It contains the name of the connection, specified in the CONNECT statement, to which you switch or fail to switch.
DISCONNECT	It contains the name of the connection, specified in the CONNECT statement, from which you disconnect or fail to disconnect. If you disconnect, and then you execute a DISCONNECT statement for a connection that is not current, the **CONNECTION_NAME** field remains unchanged.
DISCONNECT ALL	The **CONNECTION_NAME** field is blank if the statement executes successfully. If the statement does not execute successfully, the **CONNECTION_NAME** field contains the names of all the connections, specified in your CONNECT statement, from which you did not disconnect. However, this information does not mean that the connection still exists.

If the CONNECT is successful, the **CONNECTION_NAME** field is set to the following values:

- The name of the database environment as specified in the CONNECT statement if the CONNECT does not include the AS clause
- The name of the connection (identifier after the AS keyword) if the CONNECT includes the AS clause

The DATABASE Statement

When you execute a DATABASE statement, the **CONNECTION_NAME** field is blank.

Using GET DIAGNOSTICS for Error Checking

The GET DIAGNOSTICS statement returns information held in various fields of the diagnostic area. For each field in the diagnostic area that you want to access, you must supply a host variable with a compatible data type.

The following examples illustrate using the GET DIAGNOSTICS statement to display error information. The first example shows an ESQL/C error display routine called **disp_sqlstate_err()**.

```
void disp_sqlstate_err()
{
int j;

EXEC SQL BEGIN DECLARE SECTION;
    int exception_count;
    char overflow[2];
    int exception_num=1;
    char class_id[255];
    char subclass_id[255];
    char message[255];
    int messlen;
    char sqlstate_code[6];
    int i;
EXEC SQL END DECLARE SECTION;

    printf("---------------------------------");
    printf("-------------------------\n");
    printf("SQLSTATE: %s\n",SQLSTATE);
    printf("SQLCODE: %d\n", SQLCODE);
    printf("\n");

    EXEC SQL get diagnostics :exception_count = NUMBER,
        :overflow = MORE;
    printf("EXCEPTIONS:  Number=%d\t", exception_count);
    printf("More? %s\n", overflow);
    for (i = 1; i <= exception_count; i++)
    {
        EXEC SQL get diagnostics  exception :i
            :sqlstate_code = RETURNED_SQLSTATE,
            :class_id = CLASS_ORIGIN, :subclass_id = SUBCLASS_ORIGIN,
            :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
        printf("- - - - - - - - - - - - - - - - - - - - -\n");
        printf("EXCEPTION %d: SQLSTATE=%s\n", i,
            sqlstate_code);
        message[messlen-1] ='\0';
        printf("MESSAGE TEXT: %s\n", message);
```

```
        j = stleng(class_id);
        while((class_id[j] == '\0') ||
              (class_id[j] == ' '))
            j--;
        class_id[j+1] = '\0';
        printf("CLASS ORIGIN: %s\n",class_id);

        j = stleng(subclass_id);
        while((subclass_id[j] == '\0') ||
              (subclass_id[j] == ' '))
            j--;
        subclass_id[j+1] = '\0';
        printf("SUBCLASS ORIGIN: %s\n",subclass_id);
    }

    printf("---------------------------------");
    printf("-------------------------\n");
}
```

The following program shows how the GET DIAGNOSTICS statement retrieves ESQL/COBOL exception information:

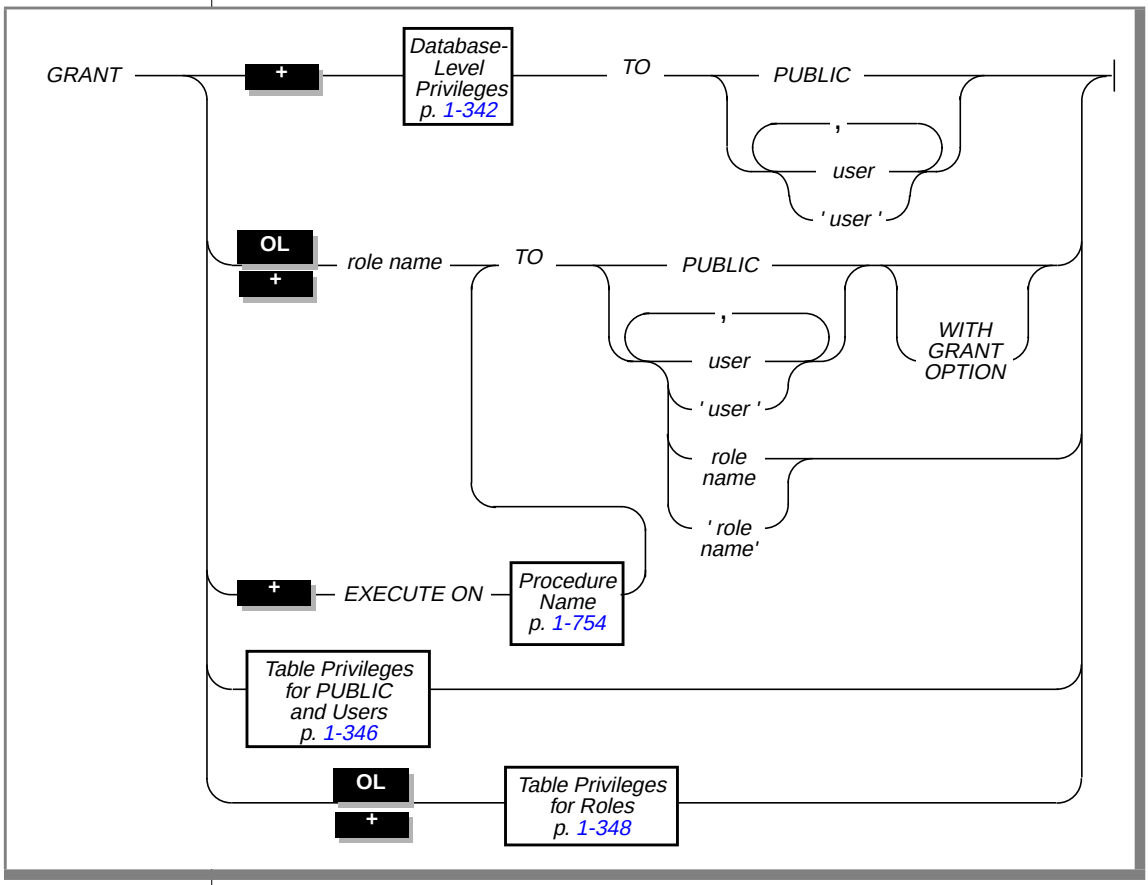
```
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID.
     DIAGCHK.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IFXSUN.
 OBJECT-COMPUTER. IFXSUN.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
*Declare variables.
*
 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 77  EXCEPTION-COUNT  PIC S9(9) COMP-5.
 77  ROW-COUNT PIC S9(9) COMP-5.
 77  MORE-CHAR PIC X(1).
 77  COUNT-EX  PIC S9(9) VALUE 1 COMP-5.
 77  SQLSTATE PIC X(5).
 77  CLASS-ORIGIN PIC X(254).
 77  SUBCLASS-ORIGIN PIC X(254).
 77  ERROR-MESS  PIC X(254).
 77  MESS-LEN  PIC S9(9) COMP-5.
 77  SERVER-NAME PIC X(254).
 77  CONNECT-NAME PIC X(254).
 EXEC SQL END DECLARE SECTION END-EXEC.
*
 PROCEDURE DIVISION.
 RESIDENT SECTION 1.
*
*Begin Main routine.  Execute an SQL statement.
*Determine number of exceptions and pass
*control to ERR-CHK subroutine to diagnose each
*exception.
*
 MAIN.
     DISPLAY 'START PROGRAM.'.
     DISPLAY 'SQLSTATE VALUE IS: ', SQLSTATE.
     DISPLAY 'SQLCODE VALUE IS: ', SQLCODE.
     DISPLAY '************************************'.
     DISPLAY 'CONNECTING TO DATABASE.'.
     DISPLAY '************************************'.
     EXEC SQL CONNECT TO "stores7" END-EXEC.
     DISPLAY 'RUNNING DIAGNOSTICS EVALUATION.'.
     DISPLAY '************************************'.
     EXEC SQL GET DIAGNOSTICS
         :EXCEPTION-COUNT=NUMBER END-EXEC.
     DISPLAY 'NUMBER OF EXCEPTIONS IS: ', EXCEPTION-COUNT.
     DISPLAY 'NUMBER OF ROWS MODIFIED?: ', ROW-COUNT.
```

```
           DISPLAY 'MORE EXCEPTIONS DETECTED?: ', MORE-CHAR.
           DISPLAY 'PERFORMING ERROR CHECKING.'.
           DISPLAY '************************************'.
           PERFORM ERR-CHK UNTIL COUNT-EX IS GREATER THAN
               EXCEPTION-COUNT.
           DISPLAY '************************************'.
           DISPLAY 'DISCONNECTING FROM DATABASE.'.
           DISPLAY '************************************'.
           EXEC SQL DISCONNECT CURRENT END-EXEC.
           DISPLAY 'END PROGRAM.'.
       STOP RUN.
      *
      *Subroutine to diagnose each exception generated by the
      *execution of an SQL CONNECT TO statement.  Display the
      *diagnostic information.
      *
       ERR-CHK.
           EXEC SQL GET DIAGNOSTICS EXCEPTION :COUNT-EX
               :SQLSTATE=RETURNED_SQLSTATE,
               :CLASS-ORIGIN=CLASS_ORIGIN,
               :SUBCLASS-ORIGIN=SUBCLASS_ORIGIN,
               :ERROR-MESS=MESSAGE_TEXT,
               :MESS-LEN=MESSAGE_LENGTH,
               :SERVER-NAME=SERVER_NAME,
               :CONNECT-NAME=CONNECTION_NAME
           END-EXEC.
           DISPLAY 'THE SQLSTATE VALUE IS: ', SQLSTATE.
           DISPLAY 'THE CLASS CODE ORIGIN IS: ', CLASS-ORIGIN.
           DISPLAY 'THE SUBCLASS CODE ORIGIN IS: ', SUBCLASS-ORIGIN.
           DISPLAY 'THE ERROR MESSAGE IS: ', ERROR-MESS.
           DISPLAY 'THE ERROR MESSAGE LENGTH IS: ', MESS-LEN.
           DISPLAY 'THE SERVER NAME IS: ', SERVER-NAME.
           DISPLAY 'THE CONNECTION NAME IS: ', CONNECT-NAME.
           ADD 1 TO COUNT-EX.
      *
```

GRANT

Use the GRANT statement to grant privileges on a database, table, view, or procedure to a user or a role. You can also use GRANT to grant privileges on a table, view, or procedure to a role. In addition, you can use GRANT to grant a role to a user or to another role.

Syntax



Element	Purpose	Restrictions	Syntax
role name	The name of the role that is granted, or the name of the role to which another role is granted	The role must have been created with the CREATE ROLE statement.	Identifier, p. 1-723
user	The name of the user to whom a role is granted, or the name of a user who receives the specified privilege	If you use quotes around *user*, the name of the user is stored exactly as you typed it. In an ANSI-compliant database, the name of the user is stored as uppercase letters if you do not use quotes around *user*. If you grant a privilege to PUBLIC, you do not need to grant the privilege to individual users because PUBLIC extends the privilege to all authorized users.	Identifier, p. 1-723

Usage

A GRANT statement can extend user privileges but cannot limit existing privileges. Later GRANT statements do not affect privileges already granted to a user. When database-level privileges collide with table-level privileges, the more-restrictive privileges take precedence. You can grant table-level privileges on a table or on a view.

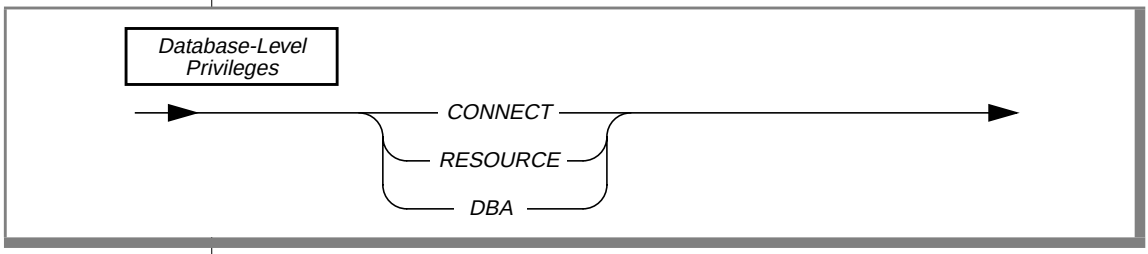
Privileges granted to users remain in effect until you cancel them with a REVOKE statement. Only grantors can revoke the privileges that they previously issued.

You can grant privileges to a role, and you can grant a role to individual users or to another role. See "Granting a Role to a User or Another Role" on page 1-345 for further information.

SE

You cannot use a ROLLBACK WORK statement to undo a GRANT statement that executes successfully. If you roll back a transaction that contains a GRANT statement, the privilege is not revoked, and you do not receive an error message. ♦

Database-Level Privileges



When you create a database, you alone have access to it. The database remains inaccessible to other users until you, as DBA, grant database privileges.

Three levels of database privileges control access. These privilege levels are, from lowest to highest, Connect, Resource, and DBA. These privileges are associated with the following keywords.

Privilege	Functions
CONNECT	Gives you the ability to query and modify data. You can modify the database schema if you own the object you want to modify. Any user with the Connect privilege can perform the following functions: ■ Connect to the database with the CONNECT statement or another connection statement ■ Execute SELECT, INSERT, UPDATE, and DELETE statements, provided the user has the necessary table-level privileges ■ Create views, provided the user has the Select privilege on the underlying tables ■ Create synonyms ■ Create temporary tables and create indexes on the temporary tables ■ Alter or drop a table or an index, provided the user owns the table or index (or has Alter, Index, or References privileges on the table) ■ Grant privileges on a table or view, provided the user owns the table (or has been given privileges on the table with the WITH GRANT OPTION keyword)
RESOURCE	Gives you the ability to extend the structure of the database. In addition to the capabilities of the Connect privilege, the holder of the Resource privilege can perform the following functions: ■ Create new tables ■ Create new indexes ■ Create new procedures

(1 of 2)

Privilege	Functions
DBA	Has all the capabilities of the Resource privilege as well as the ability to perform the following functions: ■ Grant any database-level privilege, including the DBA privilege, to another user ■ Grant any table-level privilege to another user ■ Grant any table-level privilege to a role ■ Grant a role to a user or to another role ■ Execute the SET SESSION AUTHORIZATION statement ■ Use the NEXT SIZE keyword to alter extent sizes in the system catalog ■ Insert, delete, or update rows of any system catalog table except **systables** ■ Drop any object, regardless of its owner ■ Create tables, views, and indexes, and specify another user as owner of the objects ■ Execute the DROP DATABASE statement ■ Execute the DROP DISTRIBUTIONS option of the UPDATE STATISTICS statement
SE	■ Execute the START DATABASE and ROLLFORWARD DATABASE statements ♦

(2 of 2)

User **informix** has the privilege required to alter tables in the system catalog, including the **systables** table.

Warning: *Although the user **informix** and DBAs can modify most system catalog tables (only user **informix** can modify systables), Informix strongly recommends that you do not update, delete, or alter any rows in them. Modifying the system catalog tables can destroy the integrity of the database.*

The following example uses the PUBLIC keyword to grant the Connect privilege on the currently active database to all users:

```
GRANT CONNECT TO PUBLIC
```

Granting a Role to a User or Another Role

You can use the GRANT statement to grant a role to another role or user. You can only grant roles that have been created with the CREATE ROLE statement. See the CREATE ROLE statement on page 1-145 for an explanation of roles.

After a role is granted, you must use the SET ROLE statement to enable the role. Users who have been granted a role with the WITH GRANT OPTION can grant that role to other users or roles. Roles that are granted to users remain granted until a REVOKE statement cancels them.

Table-level privileges and the Execute privilege to stored procedures can be granted to roles. Database-level privileges cannot be granted to roles.

The DBA or a user who is granted the role with the WITH GRANT OPTION can grant a role to a user or to another role. A role cannot be granted to itself, either directly or indirectly. The following statement causes an error:

```
GRANT engineer TO engineer -- Causes an error
```

The following example grants the role **engineer** to the user **maryf**:

```
GRANT engineer TO maryf
```

The following example grants the role **acct** to the role **engineer**:

```
GRANT acct TO engineer
```

The following example grants the role **engineer** with the WITH GRANT OPTION to the user **maryf**. This privilege allows **maryf** to grant the role **engineer** to other users or roles.

```
GRANT engineer TO maryf WITH GRANT OPTION
```

Stored Procedure Privileges

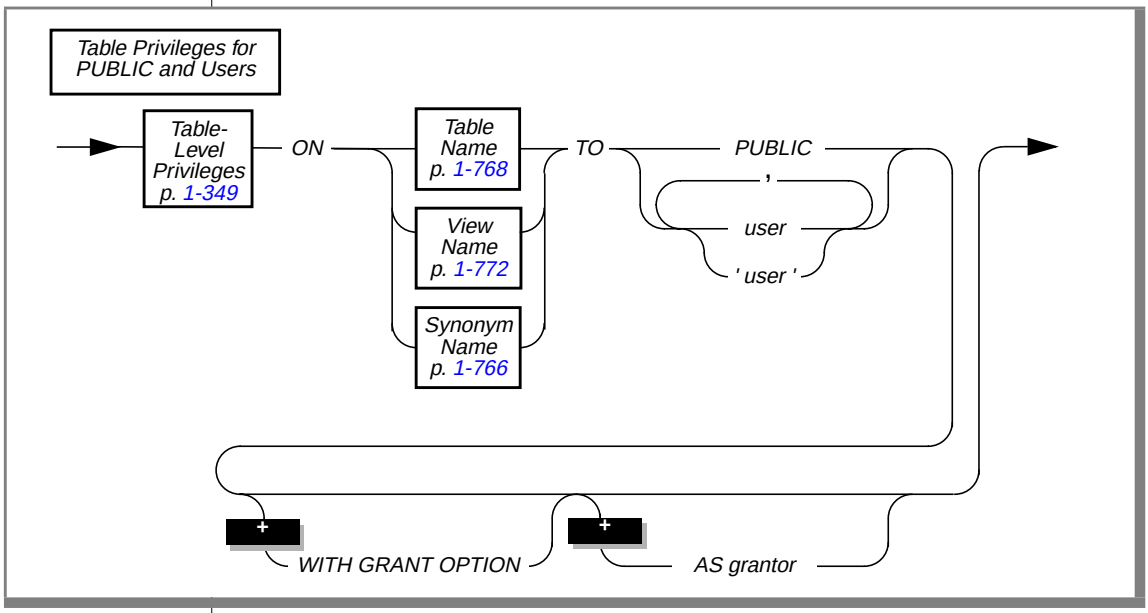
Use the EXECUTE ON option with a procedure name to grant another user or a role the ability to run a stored procedure that you own.

When you create an owner-privileged stored procedure, the default privilege is PUBLIC.

SE

If you create a procedure in a database that is ANSI compliant, no default-level privileges are granted. ♦

Table Privileges for PUBLIC and Users



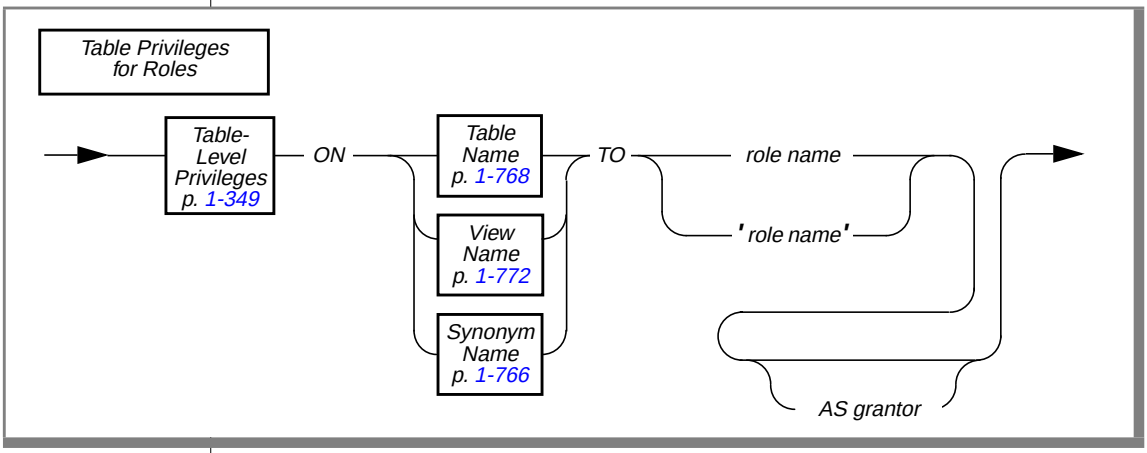
Element	Purpose	Restrictions	Syntax
grantor	The name of the person who is to be listed as the source of the specified privilege in the **systabauth** system catalog table. The person who issues the GRANT statement is the default grantor of the privilege.	If you specify someone else as the grantor of the specified privilege to *user*, you cannot later revoke that privilege from *user*.	Identifier, p. 1-723
user	The name of the user who receives the specified privilege	If you use quotes around *user*, the name of the user is stored exactly as you typed it. In an ANSI-compliant database, the name of the user is stored as uppercase letters if you do not use quotes around *user*. If you grant a privilege to PUBLIC, you do not need to grant the privilege to individual users because PUBLIC extends the privilege to all authorized users. Also see "Restricting Privileges at the Table Level" on page 1-352.	Identifier, p. 1-723

You can grant privileges on a table, view, or synonym to a user, a list of users, or all users (PUBLIC). When you grant these privileges to users or PUBLIC, you can also specify the WITH GRANT OPTION clause and the AS *grantor* clause.

The following example grants the table-level privilege Insert on **table1** to the user named **mary**:

```
GRANT INSERT ON table1 TO mary
```

Table Privileges for Roles



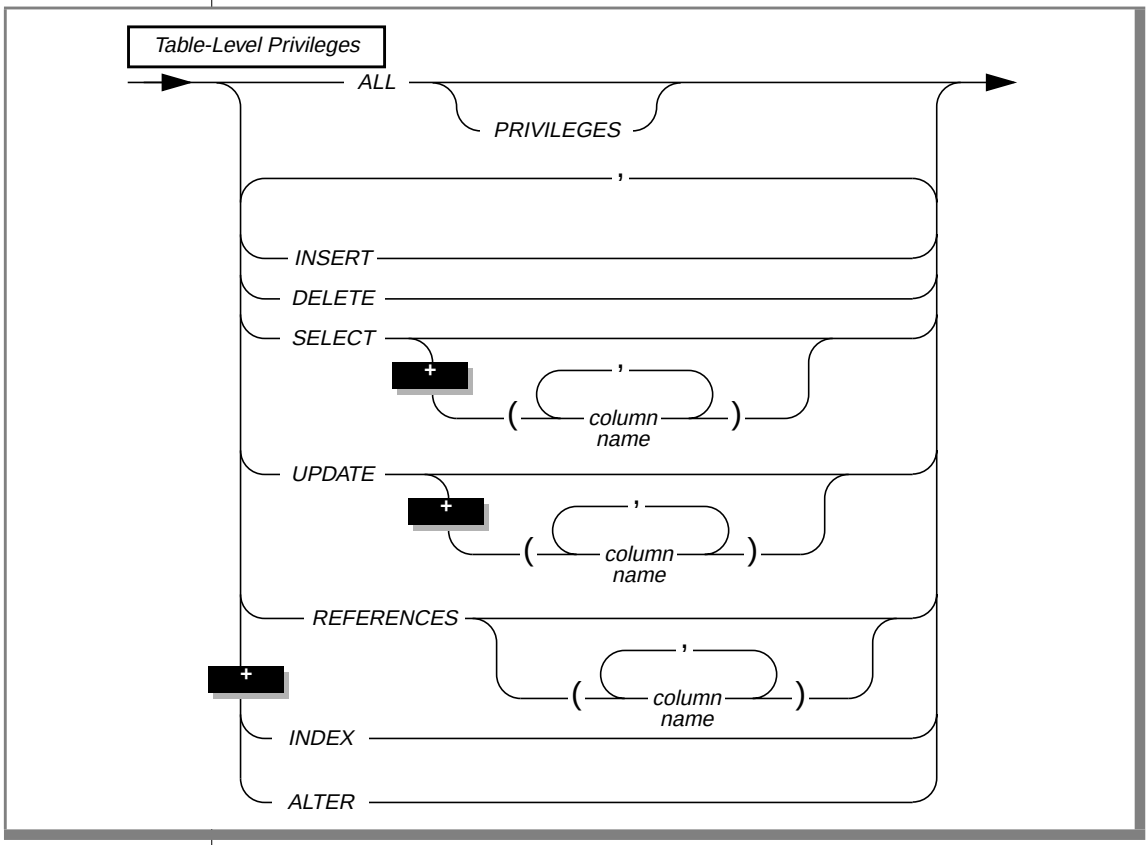
Element	Purpose	Restrictions	Syntax
grantor	The name of the person who is to be listed as the source of the specified privilege in the **systabauth** system catalog table. The person who issues the GRANT statement is the default grantor of the privilege.	If you specify someone else as the grantor of the specified privilege to *role name*, you cannot later revoke that privilege from *role name*.	Identifier, p. 1-723
role name	The name of the role to which the specified privilege is granted	The role must have been created with the CREATE ROLE statement.	Identifier, p. 1-723

You can grant privileges on a table, view, or synonym to a role. When you grant these privileges to a role, you can also specify the AS *grantor* clause, but you cannot specify the WITH GRANT OPTION clause.

The following example grants the table-level privilege Insert on **table1** to the role **engineer**:

```
GRANT INSERT ON table1 TO engineer
```

Table-Level Privileges



Element	Purpose	Restrictions	Syntax
column name	The name of the column or columns to which a Select, Update, or References privilege is restricted. If you omit *column name*, the privilege applies to all columns in the specified table.	The specified column or columns must exist.	Identifier, p. 1-723

As the owner of a table, or as the DBA, you control access to the table through seven table-level privileges. Four privileges control access to the table data: Select, Insert, Delete, and Update. The remaining three privileges are Index, which controls index creation; Alter, which controls the ability to change the table definition or alter an index; and References, which controls the ability to place referential constraints on table columns.

The person who creates a table is its owner and receives all seven table-level privileges. Table ownership cannot be transferred to another user.

To use the GRANT statement, list the privileges that you are granting to *user*. If you are granting all table-level privileges, use the keyword ALL. If you are granting the Select, Update, or References privilege, you can limit the privileges by listing the names of specific columns.

If you are granting the Alter privilege with the intent of allowing a user to make changes to a table, you must also grant the Resource privilege for the database in which the table resides.

If you are granting the Index privilege with the intent of allowing *user* to make changes to the underlying structure of a table, be aware that *user* must also have the Resource privilege for the database to be able to modify the database structure. The table-level privileges are defined in the following table.

Privilege	Functions
INSERT	Provides the ability to insert rows.
DELETE	Provides the ability to delete rows.
SELECT	Provides the ability to name any column in SELECT statements. You can restrict the Select privilege to one or more columns by listing them.
UPDATE	Provides the ability to name any column in UPDATE statements. You can restrict the Update privilege to one or more columns by listing them.

(1 of 2)

Privilege	Functions
REFERENCES	Provides the ability to reference columns in referential constraints. You must have the Resource privilege to take advantage of the References privilege. (However, you can add a referential constraint during an ALTER TABLE statement. This action does not require that you have the Resource privilege on the database.) You can restrict the References privilege to one or more columns by listing them. You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to place cascading deletes on a table.
INDEX	Provides the ability to create permanent indexes. You must have Resource privilege to use the Index privilege. (Any user with the Connect privilege can create an index on temporary tables.)
ALTER	Provides the ability to add or delete columns, modify column data types, or add or delete constraints. This privilege also provides the ability to set the object mode of unique indexes and constraints to the enabled, disabled, or filtering mode. In addition, this privilege provides the ability to set the object mode of nonunique indexes and triggers to the enabled or disabled modes. You must have Resource privilege to use the Alter privilege.
ALL	Provides all privileges. The PRIVILEGES keyword is optional.

(2 of 2)

The following example grants, to users **mary** and **john,** the Delete and Select privileges on all columns. It also grants the Update privilege on **customer_num**, **fname**, and **lname** for the **customer** table.

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
    ON customer TO mary, john
```

To grant these table-level privileges to all authorized users, use the keyword PUBLIC as shown in the following example:

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
    ON customer TO PUBLIC
```

Restricting Privileges at the Table Level

You must take action to restrict privileges at the table level. The database server automatically grants to PUBLIC all table-level privileges, except Alter and References, when you create a table. To limit table access, you must revoke all privileges and regrant only those you want, as the following example shows:

```
REVOKE ALL ON customer FROM PUBLIC
GRANT ALL ON customer TO john, mary
GRANT SELECT (fname, lname, company, city)
    ON customer TO PUBLIC
```

ANSI

In an ANSI-compliant database, only the table owner receives privileges when a table is created. ♦

Behavior of the ALL Keyword

The ALL keyword keyword grants all table-level privileges to the specified user. If any or all of the table-level privileges do not exist for the grantor, the GRANT statement with the ALL keyword succeeds, but the following SQLSTATE warning is returned:

```
01007 - Privilege not granted.
```

For example, assume that the user **ted** has the Select and Insert privileges on the **customer** table with the authority to grant those privileges to other users. User **ted** wants to grant all seven table-level privileges to user **tania**. So user **ted** issues the following GRANT statement:

```
GRANT ALL ON customer TO tania
```

This statement executes successfully but returns SQLSTATE code 01007. The SQLSTATE warning is returned with a successful statement for the following reasons:

- The statement succeeds in granting the Select and Insert privileges to user **tania** because user **ted** has those privileges and the right to grant those privileges to other users.
- SQLSTATE code 01007 is returned because the other five privileges implied by the ALL keyword (the Delete, Update, References, Index, and Alter privileges) were not grantable by user **ted** and, therefore, were not granted to user **tania**.

WITH GRANT OPTION Keyword

Using the WITH GRANT OPTION keyword conveys the specified privilege to *user* along with the right to grant those same privileges to other users. You create a chain of privileges that begins with you and extends to *user* as well as to whomever *user* conveys the right to grant privileges. If you use the WITH GRANT OPTION keyword, you can no longer control the dissemination of privileges.

If you revoke from *user* the privilege that you granted using the WITH GRANT OPTION keyword, you sever the chain of privileges. That is, when you revoke privileges from *user*, you automatically revoke the privileges of all users who received privileges from *user* or from the chain that *user* created (unless *user*, or the users who received privileges from *user*, were granted the same set of privileges by someone else). The following examples illustrate this situation. You, as the owner of the table **items**, issue the following statements to grant access to the user **mary**:

```
REVOKE ALL ON items FROM PUBLIC
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION
```

The user **mary** uses her new privilege to grant users **cathy** and **paul** access to the table.

```
GRANT SELECT, UPDATE ON items TO cathy
GRANT SELECT ON items TO paul
```

Later you issue the following statement to cancel access privileges for the user **mary** on the **items** table:

```
REVOKE SELECT, UPDATE ON items FROM mary
```

This single statement effectively revokes all privileges on the **items** table from the users **mary**, **cathy**, and **paul**.

If you want to create a chain of privileges with another user as the source of the privilege, use the AS *grantor* clause.

AS grantor Clause

The AS *grantor* clause lets you establish a chain of privileges with another user as the source of the privileges. This relinquishes your ability to break the chain of privileges. Even a DBA cannot revoke a privilege unless that DBA originally granted the privilege. The following example illustrates this situation. You are the owner of the **items** table, and you grant all privileges to the user **tom**, along with the right to grant all privileges:

```
REVOKE ALL ON items FROM PUBLIC
GRANT ALL ON items TO tom WITH GRANT OPTION
```

The following example illustrates a different situation. You also grant Select and Update privileges to the user **jim**, but you specify that the grant is made as the user **tom**. (The records of the database server show that the user **tom** is the grantor of the grant in the **systabauth** system catalog table, rather than you.)

```
GRANT SELECT, UPDATE ON items TO jim AS tom
```

Later, you decide to revoke privileges on the **items** table from the user **tom**, so you issue the following statement:

```
REVOKE ALL ON items FROM tom
```

When you try to revoke privileges from the user **jim** with a similar statement, however, the database server returns an error, as the following example shows:

```
REVOKE SELECT, UPDATE ON items FROM jim

580: Cannot revoke permission.
```

You get an error because the database-server record shows the original grantor as the user **tom**, and you cannot revoke the privilege. Although you are the table owner, you cannot revoke a privilege that another user granted.

Privileges on a View

You must explicitly grant access privileges on the view to users, because no automatic grant is made to **public**, as is the case with a newly created table.

When you create a view, if you do not own the underlying tables, you must have at least the Select privilege on the table or columns. As view creator, the privileges you have on the underlying table apply to the view built on the table. You do not receive any other privileges or the ability to grant any other privileges because you own the view on the table. If the view meets all the requirements for updating, any Delete, Insert, or Update privileges that you have on the table also apply to the view.

You can grant (or revoke) privileges on a view only if you are the owner of the underlying tables or if you received these privileges on the table with the right to grant them (the WITH GRANT OPTION keyword). You cannot grant Index, Alter, or References privileges on a view (or the All privilege because All includes Index, References, and Alter).

For views that reference only tables in the current database, if the owner of a view loses the Select privilege on any table underlying the view, the view is dropped.

For detailed information, refer to the CREATE TABLE statement, which also describes creating views.

References

See the CREATE TABLE, GRANT FRAGMENT, REVOKE, and REVOKE FRAGMENT statements in this manual.

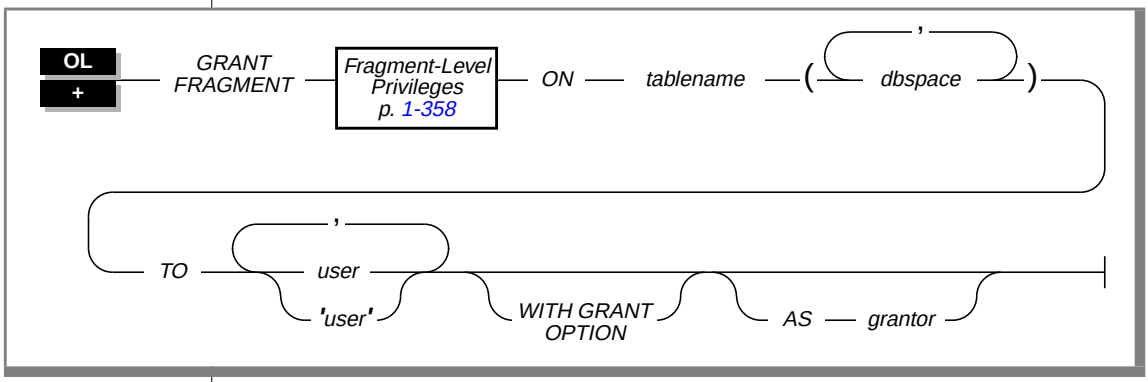
For information on roles, see the CREATE ROLE, DROP ROLE, and SET ROLE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussions of database-level privileges and table-level privileges in Chapter 4 and the discussion of privileges and security in Chapter 10.

GRANT FRAGMENT

The GRANT FRAGMENT statement enables you to grant Insert, Update, and Delete privileges on individual fragments of a fragmented table.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace where the fragment is stored. Use this parameter to specify the fragment or fragments on which privileges are to be granted. There is no default value.	You must specify at least one dbspace. The specified dbspaces must exist.	Identifier, p. 1-723
grantor	The name of the user who is to be listed as the grantor of the specified privileges in the **grantor** column of the **sysfragauth** system catalog table. The user who issues the GRANT FRAGMENT statement is the default grantor of the privileges.	The user specified in *grantor* must be a valid user.	Identifier, p. 1-723

(1 of 2)

Element	Purpose	Restrictions	Syntax
tablename	The name of the table that contains the fragment or fragments on which privileges are to be granted. There is no default value.	The specified table must exist and must be fragmented by expression.	Table Name, p. 1-768
user	The name of the user or users to whom the specified privileges are to be granted. There is no default value.	If you put quotes around *user*, the name of the user is stored exactly as you typed it. In an ANSI-compliant database, the name of the user is stored as uppercase letters if you do not use quotes around *user*.	Identifier, p. 1-723

(2 of 2)

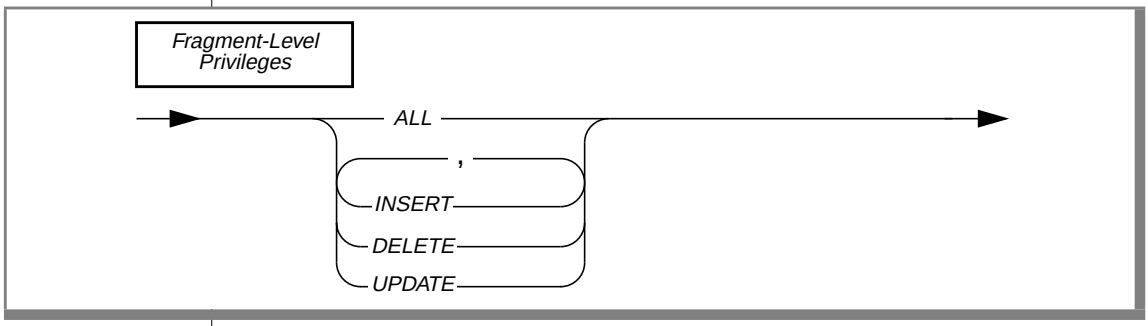
Usage

The GRANT FRAGMENT statement is similar to the GRANT statement. Both statements grant privileges to users. The difference between the two statements is that you use GRANT to grant privileges on a table while you use GRANT FRAGMENT to grant privileges on table fragments.

Use the GRANT FRAGMENT statement to grant the Insert, Update, or Delete privilege on one or more fragments of a table to one or more users.

The GRANT FRAGMENT statement is valid only for tables that are fragmented according to an expression-based distribution scheme. For an explanation of expression-based distribution schemes, see the ALTER FRAGMENT statement on page 1-22.

Fragment-Level Privileges



The following table defines each of the fragment-level privileges.

Privilege	Functions
ALL	Grants Insert, Update, and Delete privileges on a table fragment.
INSERT	Grants Insert privilege on a table fragment. This privilege gives the user the ability to insert rows in the fragment.
DELETE	Grants Delete privilege on a table fragment. This privilege gives the user the ability to delete rows in the fragment.
UPDATE	Grants Update privilege on a table fragment. This privilege gives the user the ability to update rows in the fragment and to name any column of the table in an UPDATE statement.

Definition of Fragment-Level Authority

When a fragmented table is created in an ANSI-compliant database, the table owner implicitly receives all table-level privileges on the new table, but no other users receive privileges.

When a fragmented table is created in a database that is not ANSI compliant, the table owner implicitly receives all table-level privileges on the new table, and other users (that is, PUBLIC) receive the following default set of privileges on the table: Select, Update, Insert, Delete, and Index. The privileges granted to PUBLIC are explicitly recorded in the **systabauth** system catalog table.

A user who has table privileges on a fragmented table has the privileges implicitly on all fragments of the table. These privileges are not recorded in the **sysfragauth** system catalog table.

Whether or not the database is ANSI compliant, you can use the GRANT FRAGMENT statement to grant explicit Insert, Update, and Delete privileges on one or more fragments of a table that is fragmented by expression. The privileges granted by the GRANT FRAGMENT statement are explicitly recorded in the **sysfragauth** system catalog table.

The Insert, Update, and Delete privileges that are conferred on table fragments by the GRANT FRAGMENT statement are collectively known as fragment-level privileges or fragment-level authority.

Role of Fragment-Level Authority in Command Validation

Fragment-level authority lets users execute INSERT, DELETE, and UPDATE statements on table fragments even if they lack Insert, Update, and Delete privileges on the table as a whole. Users who lack privileges at the table level can insert, delete, and update rows in authorized fragments because of the algorithm by which OnLine validates commands. This algorithm consists of the following checks:

1. When a user executes an INSERT, DELETE, or UPDATE statement, the database server first checks whether the user has the table authority necessary for the operation attempted. If the table authority exists, the command continues processing.
2. If the table authority does not exist, the database server checks whether the table is fragmented by expression. If the table is not fragmented by expression, the database server returns an error to the user. This error indicates that the user does not have the privilege to execute the command.

3. If the table is fragmented by expression, the database server checks whether the user has the fragment authority necessary for the operation attempted. If the fragment authority exists, the command continues processing. If the fragment authority does not exist, the database server returns an error to the user. This error indicates that the user does not have the privilege to execute the command.

Duration of Fragment-Level Authority

The duration of fragment-level authority is tied to the duration of the fragmentation strategy for the table as a whole.

If you drop a fragmentation strategy by means of a DROP TABLE statement or the INIT, DROP, or DETACH clauses of an ALTER FRAGMENT statement, you also drop any authorities that exist for the affected fragments. Similarly, if you drop a dbspace, you also drop any authorities that exist for the fragment that resides in that dbspace.

Tables that are created as a result of a DETACH or INIT clause of an ALTER FRAGMENT statement do not keep the authorities that the former fragment or fragments had when they were part of the fragmented table. Instead, such tables assume the default table authorities.

If a table with fragment authorities defined on it is changed to a table with a round-robin strategy or some other expression strategy, the fragment authorities are also dropped, and the table assumes the default table authorities.

Granting Privileges on One Fragment or a List of Fragments

You can grant fragment-level privileges on one fragment of a table or on a list of fragments.

Granting Privileges on One Fragment

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp1** to the user **larry**:

```
GRANT FRAGMENT ALL ON customer (dbsp1) TO larry
```

Granting Privileges on More Than One Fragment

The following statement grants the Insert, Update, and Delete privileges on the fragments of the **customer** table in **dbsp1** and **dbsp2** to the user **millie**:

```
GRANT FRAGMENT ALL ON customer (dbsp1, dbsp2) TO millie
```

Granting Privileges on All Fragments of a Table

If you want to grant privileges on all fragments of a table to the same user or users, you can use the GRANT statement instead of the GRANT FRAGMENT statement. However, you can also use the GRANT FRAGMENT statement for this purpose.

Assume that the **customer** table is fragmented by expression into three fragments, and these fragments reside in the dbspaces named **dbsp1**, **dbsp2**, and **dbsp3**. You can use either of the following statements to grant the Insert privilege on all fragments of the table to the user **helen**:

```
GRANT FRAGMENT INSERT ON customer (dbsp1, dbsp2, dbsp3)
TO helen;

GRANT INSERT ON customer TO helen;
```

Granting Privileges to One User or a List of Users

You can grant fragment-level privileges to a single user or to a list of users.

Granting Privileges to One User

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp3** to the user **oswald**:

```
GRANT FRAGMENT ALL ON customer (dbsp3) TO oswald
```

Granting Privileges to a List of Users

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp3** to the users **jerome** and **hilda**:

```
GRANT FRAGMENT ALL ON customer (dbsp3) TO jerome, hilda
```

Granting One Privilege or a List of Privileges

When you specify fragment-level privileges in a GRANT FRAGMENT statement, you can specify one privilege, a list of privileges, or all privileges.

Granting One Privilege

The following statement grants the Update privilege on the fragment of the **customer** table in **dbsp1** to the user **ed**:

```
GRANT FRAGMENT UPDATE ON customer (dbsp1) TO ed
```

Granting a List of Privileges

The following statement grants the Update and Insert privileges on the fragment of the **customer** table in **dbsp1** to the user **susan**:

```
GRANT FRAGMENT UPDATE, INSERT ON customer (dbsp1) TO susan
```

Granting All Privileges

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **dbsp1** to the user **harry**:

```
GRANT FRAGMENT ALL ON customer (dbsp1) TO harry
```

WITH GRANT OPTION Clause

By including the WITH GRANT OPTION clause in the GRANT FRAGMENT statement, you convey the specified fragment-level privileges to a user and the right to grant those same privileges to other users.

The following statement grants the Update privilege on the fragment of the **customer** table in **dbsp3** to the user **george** and gives this user the right to grant the Update privilege on the same fragment to other users:

```
GRANT FRAGMENT UPDATE ON customer (dbsp3) TO george
    WITH GRANT OPTION
```

AS grantor Clause

The AS *grantor* clause is optional in a GRANT FRAGMENT statement. Use this clause to specify the grantor of the privilege.

Including the AS grantor Clause

When you include the AS *grantor* clause in the GRANT FRAGMENT statement, you specify that the user who is named in the *grantor* parameter is listed as the grantor of the privilege in the **grantor** column of the **sysfragauth** system catalog table.

In the following example, the DBA grants the Delete privilege on the fragment of the **customer** table in **dbsp3** to the user **martha**. In the GRANT FRAGMENT statement, the DBA uses the AS *grantor* clause to specify that the user **jack** is listed as the grantor of the privilege in the **sysfragauth** system catalog table.

```
GRANT FRAGMENT DELETE ON customer (dbsp3) TO martha AS jack
```

Omitting the AS grantor Clause

When a GRANT FRAGMENT statement does not include the AS *grantor* clause, the user who issues the statement is the default grantor of the privileges that are specified in the statement.

In the following example, the user grants the Update privilege on the fragment of the **customer** table in **dbsp3** to the user **fred**. Because this statement does not specify the AS *grantor* clause, the user who issues the statement is listed by default as the grantor of the privilege in the **sysfragauth** system catalog table.

```
GRANT FRAGMENT UPDATE ON customer (dbsp3) TO fred
```

Consequences of the AS grantor Clause

If you omit the AS *grantor* clause, or if you specify your own user name in the *grantor* parameter, you can later revoke the privilege that you granted to the specified user. However, if you specify someone other than yourself as the grantor of the specified privilege to the specified user, only that grantor can revoke the privilege from the user.

For example, if you grant the Delete privilege on the fragment of the **customer** table in **dbsp3** to user **martha** but specify user **jack** as the grantor of the privilege, user **jack** can revoke that privilege from user **martha**, but you cannot revoke that privilege from user **martha**.

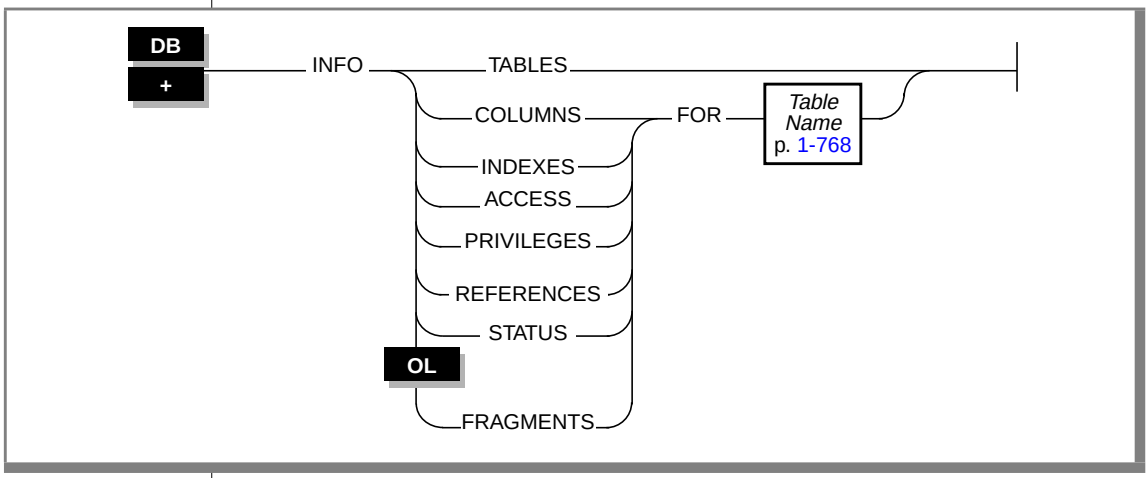
References

See the GRANT and REVOKE FRAGMENT statements in this manual.

INFO

Use the INFO statement to display a variety of information about databases and tables.

Syntax



Usage

You can display the following types of information when you issue the INFO statement:

- Tables names in the current database
- Column information for a specified table
- Index information for a specified table
- Access privileges for a specified table
- References privileges for the columns of a specified table
- Status information for a specified table
- Fragmentation strategy for a table

Instead of using the INFO statement, you can use the Info options on the <vk>SQL menu or the TABLE menu to display the same and additional information.

Displaying Tables, Columns, Indexes, and Fragments

You can use keywords in your INFO statement to display a list of tables, information about the columns of a table, information about the indexes of a table, or information about fragments.

TABLES Keyword

Use the TABLES keyword to display a list of the tables in the current database. The name of a table can appear in one of the following ways:

- If you are the owner of the **cust_calls** table, it appears as **cust_calls**.
- If you are *not* the owner of the **cust_calls** table, the owner's name precedes the table name, such as '**june'.cust_calls**.

COLUMNS Keyword

Use the COLUMNS keyword to display the names and data types of the columns in a specified table and whether null values are allowed. The following examples show an INFO statement and the resulting display of information about the columns in a table:

INFO statement requesting column information

```
INFO COLUMNS FOR cust_calls
```

Display of column information

```
Column name           Type                     Nulls

customer_num       INTEGER                      no
call_dtime         DATETIME YEAR TO MINUTE      yes
user_id            CHAR(18)                     yes
call_code          CHAR(1)                      yes
call_descr         CHAR(240)                    yes
res_dtime          DATETIME YEAR TO MINUTE      yes
res_descr          CHAR(240)                    yes
```

INDEXES Keyword

Use the INDEXES keyword to display the name, owner, and type of each index in a specified table, whether the index is clustered, and the names of the columns that are indexed. The following examples show an INFO statement and the resulting display of information about the indexes of a table:

INFO statement requesting index information

```
INFO INDEXES FOR cust_calls
```

Display of index information

```
Index name      Owner     Type       Cluster    Columns

c_num_dt_ix     velma     unique     No         customer_num
                                                call_dtime
c_num_cus_ix    velma     dupls      No         customer_num
```

FRAGMENTS Keyword

Use the FRAGMENTS keyword to display the dbspace names where fragments are located for a specified table. The following examples show an INFO statement and the resulting display of fragments for a table that is fragmented with a round-robin distribution scheme. An INFO statement that is executed on a table that is fragmented with an expression-based distribution scheme would show the expressions and the dbspaces.

INFO statement requesting fragment information

```
INFO FRAGMENTS FOR new_accts
```

Display of fragment information

```
dbsp1

dbsp2

dbsp3
```

Displaying Privileges, References, and Status

You can use keywords in your INFO statement to display information about the access privileges (including the References privilege) or the status of a table.

ACCESS Keyword

Use the ACCESS or PRIVILEGES keywords to display user access privileges for a specified table. The following examples show an INFO statement and the resulting display of user privileges for a table:

INFO statement requesting privileges information

```
INFO PRIVILEGES FOR cust_calls
```

Display of privileges information

```
User     Select            Update            Insert  Delete  Index  Alter

public    All               All               Yes     Yes     Yes    No
```

REFERENCES Keyword

Use the REFERENCES keyword to display the References privilege for users for the columns of a specified table. The following examples show an INFO statement and the resulting display:

INFO statement requesting References privilege information

```
INFO REFERENCES FOR newtable
```

Display of References privilege information

```
User         Column References

betty        col1
             col2
             col3
wilma        All
public       None
```

The output indicates that the user **betty** can reference columns **col1**, **col2**, and **col3** of the specified table; the user **wilma** can reference all the columns in the table; and **public** cannot access any columns in the table.

If you want information about database-level privileges, you must use a SELECT statement to access the **sysusers** system catalog table.

See the GRAPH and REVOKE statements for more information about database and table-access privileges.

STATUS Keyword

Use the STATUS keyword to display information about the owner, row length, number of rows and columns, creation date, and status of audit trails for a specified table. The following example displays status information for a table on an INFORMIX-SE database server:

INFO statement requesting status information

```
INFO STATUS FOR cust_calls
```

Display of status information

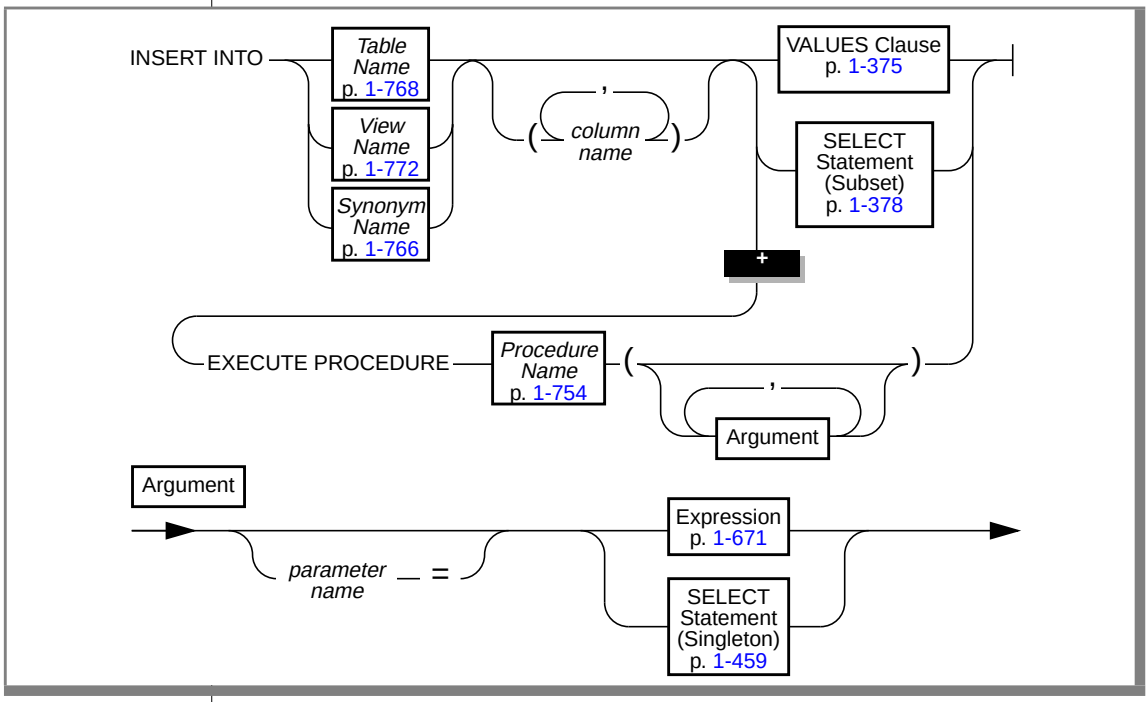
```
Table Name         cust_calls
Owner              velma
Row Size           517
Number of Rows     7
Number of Columns  7
Date Created       01/28/1993
```

The audit-trail file line does not appear for tables in the INFORMIX-OnLine Dynamic Server databases.

INSERT

Use the INSERT statement to insert one or more new rows into a table or view.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column that receives a new column value, or a list of columns that receive new values. If you specify a column list, values are inserted into columns in the order in which you list the columns. If you do not specify a column list, values are inserted into columns in the column order that was established when the table was created or last altered.	The number of columns you specify must equal the number of values supplied in the VALUES clause or by the SELECT statement, either implicitly or explicitly. If you omit a column from the column list, and the column does not have a default value associated with it, the database server places a null value in the column when the INSERT statement is executed.	Identifier, p. 1-723
parameter name	The name of an input parameter to the procedure	The input parameter must have been defined in the CREATE PROCEDURE statement for the specified procedure.	Expression, p. 1-671

Usage

Use the INSERT statement to create either a single new row of column values or a group of new rows using data selected from other tables.

To insert data into a table, you must either own the table or have the Insert privilege for the table (see the GRANT statement on page 1-340). To insert data into a view, you must have the required Insert privilege, and the view must meet the requirements explained in "Inserting Rows Through a View."

If you insert data into a table that has data integrity constraints associated with it, the inserted data must meet the constraint criteria. If it does not, the database server returns an error.

If you are using effective checking, and the checking mode is set to IMMEDIATE, all specified constraints are checked at the end of each INSERT statement. If the checking mode is set to DEFERRED, all specified constraints are *not* checked until the transaction is committed.

Specifying Columns

If you do not explicitly specify one or more columns, data is inserted into columns using the column order that was established when the table was created or last altered. The column order is listed in the **syscolumns** system catalog table.

ESQL

You can use the DESCRIBE statement with an INSERT statement to determine the column order and the data type of the columns in a table. (For more information about the DESCRIBE statement, see page 1-255.) ♦

The number of columns specified in the INSERT INTO clause must equal the number of values supplied in the VALUES clause or by the SELECT statement, either implicitly or explicitly. If you specify columns, the columns receive data in the order in which you list them. The first value following the VALUES keyword is inserted into the first column listed, the second value is inserted into the second column listed, and so on.

Inserting Rows Through a View

You can insert data through a *single-table* view if you have the Insert privilege on the view. To do this, the defining SELECT statement can select from only one table, and it cannot contain any of the following components:

- DISTINCT keyword
- GROUP BY clause
- Derived value (also referred to as a virtual column)
- Aggregate value

Columns in the underlying table that are unspecified in the view receive either a default value or a null value if no default is specified. If one of these columns does not specify a default value, and a null value is not allowed, the insert fails.

You can use data-integrity constraints to prevent users from inserting values into the underlying table that do not fit the view-defining SELECT statement. For further information, refer to the WITH CHECK OPTION discussion under the CREATE VIEW statement on page 1-227.

If several users are entering sensitive information into a single table, the USER function can limit their view to only the specific rows that each user inserted. The following example contains a view and an INSERT statement that achieve this effect:

```
CREATE VIEW salary_view AS
    SELECT lname, fname, current_salary
        FROM salary
        WHERE  entered_by = USER

INSERT INTO salary
    VALUES ('Smith', 'Pat', 75000, USER)
```

Inserting Rows with a Cursor

ESQL

If you associate a cursor with an INSERT statement, you must use the OPEN, PUT, and CLOSE statements to carry out the INSERT operation. For databases that have transactions but are not ANSI-compliant, you must issue these statements within a transaction.

If you are using a cursor that is associated with an INSERT statement, the rows are buffered before they are written to the disk. The insert buffer is flushed under the following conditions:

- The buffer becomes full.
- A FLUSH statement executes.
- A CLOSE statement closes the cursor.
- In a database that is not ANSI-compliant, an OPEN statement implicitly closes and then reopens the cursor.
- A COMMIT WORK statement ends the transaction.

When the insert buffer is flushed, the client processor performs appropriate data conversion before it sends the rows to the database server. When the database server receives the buffer, it begins to insert the rows one at a time into the database. If an error is encountered while the database server inserts the buffered rows into the database, any buffered rows following the last successfully inserted rows are discarded. ♦

Inserting Rows into a Database Without Transactions

If you are inserting rows into a database without transactions, you must take explicit action to restore inserted rows after a failure. For example, if the INSERT statement fails after inserting some rows, the successfully inserted rows remain in the table. You cannot recover automatically from a failed insert.

Inserting Rows into a Database with Transactions

If you are inserting rows into a database with transactions, and you are using explicit transactions, use the ROLLBACK WORK statement to undo the insertion. If you do not execute BEGIN WORK before the insert, and the insert fails, the database server automatically rolls back any database modifications made since the beginning of the insert.

ANSI

If you are inserting rows into an ANSI-compliant database, transactions are implicit, and all database modifications take place within a transaction. In this case, if an INSERT statement fails, use the ROLLBACK WORK statement to undo the insertions.

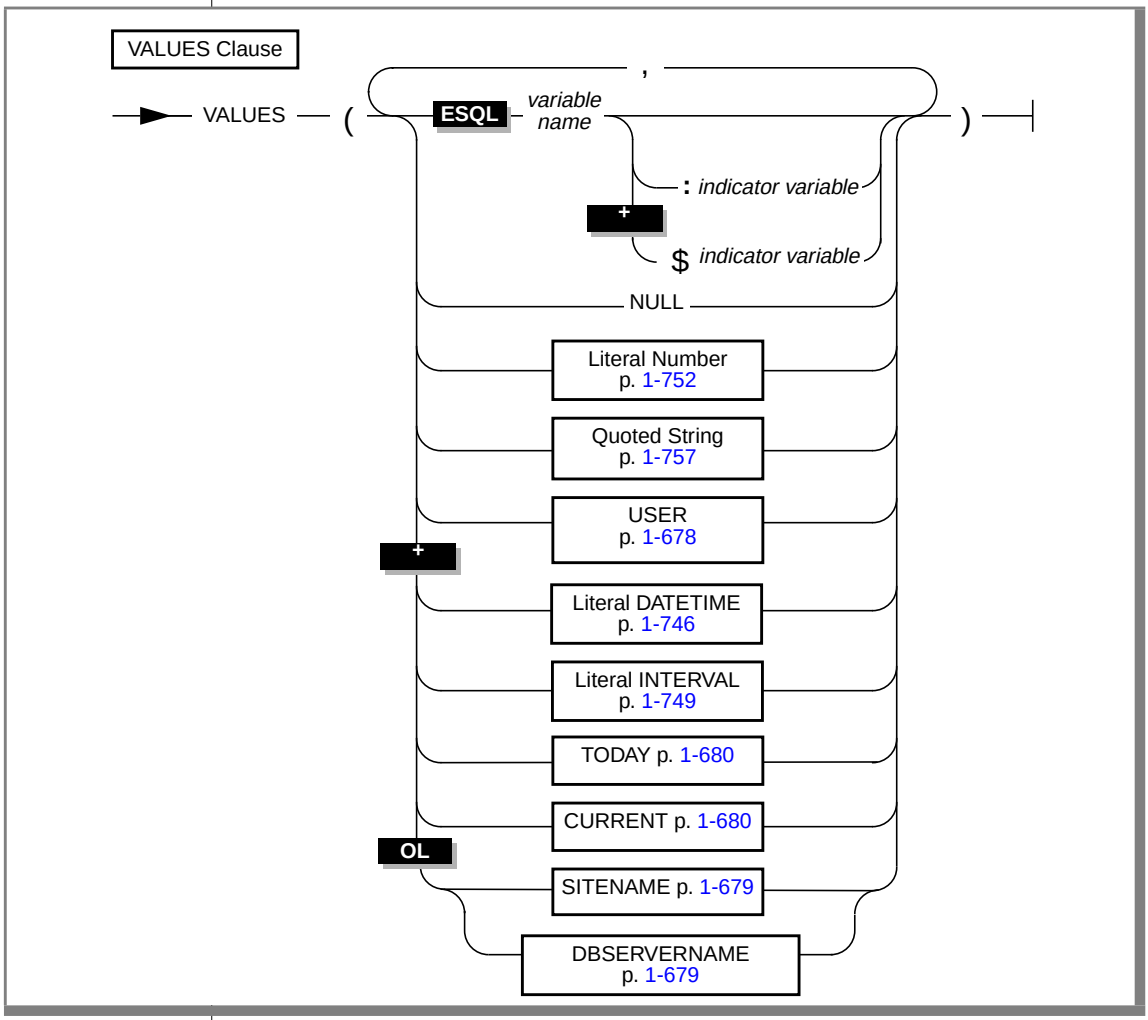
If you are using INFORMIX-OnLine Dynamic Server within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

Rows that you insert within a transaction remain locked until the end of the transaction. The end of a transaction is either a COMMIT WORK statement, where all modifications are made to the database, or a ROLLBACK WORK statement, where none of the modifications are made to the database. If many rows are affected by a *single* INSERT statement, you can exceed the maximum number of simultaneous locks permitted. To prevent this situation, either insert fewer rows per transaction or lock the page, or the entire table, before you execute the INSERT statement.

SE

To prevent this situation, either insert fewer rows per transaction, or lock the entire table before you execute the INSERT statement. ♦

VALUES Clause



Element	Purpose	Restrictions	Syntax
indicator variable	A program variable associated with *variable name* that indicates when an <vk>SQL API statement returns a null value to *variable name*	See your SQL API manual for the restrictions that apply to indicator variables in a particular language.	The name of the indicator variable must conform to language-specific rules for naming indicator variables.
variable name	A host variable that specifies a value to be inserted into a column	You can specify in *variable name* any other value option listed in the VALUES clause (NULL, Literal Number, and so on). If you specify a quoted string in *variable name*, the string can be longer than the 256-byte maximum that applies to your specified quoted strings.	The name of the host variable must conform to language-specific rules for variable names.

When you use the VALUES clause, you can insert only one row at a time. Each value that follows the VALUES keyword is assigned to the corresponding column listed in the INSERT INTO clause (or in column order if a list of columns is not specified).

If you are inserting a quoted string into a column, the maximum length of the string is 256 bytes. If you insert a value greater than 256, the database server returns an error.

ESQL

If you are using variables, you can insert quoted strings longer than 256 bytes into a table. ♦

Value and Column Data Type Compatibility

Although the values you insert do not have to be the same data type as the columns receiving them, the value data type and column data type must be compatible. You can insert only characters into CHAR columns and only numbers or characters that represent number data into number columns. The following example inserts values into the columns of the **customer** table:

```
INSERT INTO customer
    VALUES (0, 'Nadia', 'Broadam', 'Ski & Stuff',
            '89 Coniston Road', NULL, 'Short Hills',
            'NJ', '07079', '201-457-4100')
```

The database server makes every effort to perform data conversion. If the data cannot be converted, the INSERT operation fails. Data conversion also fails if the target data type cannot hold the value that is specified. For example, you cannot insert the integer 123456 into a column defined as a SMALLINT data type because this data type cannot hold a number that large.

Inserting Values into SERIAL Columns

If you want to insert consecutive serial values into a SERIAL column in the table, enter a zero for a SERIAL column in the INSERT statement. When a SERIAL column is set to zero, the database server assigns the next highest value. If you want to enter an explicit value into a SERIAL column, specify the nonzero value after you first verify that the value does not duplicate one already in the table. If the SERIAL column is uniquely indexed or has a unique constraint, and you try to insert a value that duplicates one already in the table, an error occurs. For more information about the SERIAL data type, see Chapter 3 of the *Informix Guide to SQL: Reference*.

Using Functions in the VALUES Clause

You can insert the current date, date and time, login name of the current user, or database server name of the current INFORMIX-OnLine Dynamic Server database into a column. The TODAY keyword returns the system date. The CURRENT keyword returns the system date and time. The USER keyword returns an eight-character string that contains the login account name of the current user. The SITENAME or DBSERVERNAME keyword returns the database server name where the current database resides. The following example uses the CURRENT and USER keywords to insert a new row into the **cust_calls** table:

```
INSERT INTO cust_calls (customer_num, call_dtime, user_id,
                        call_code, call_descr)
    VALUES (212, CURRENT, USER, 'L', '2 days')
```

Inserting Nulls with the VALUES Clause

When you execute an INSERT statement, a null value is inserted into any column for which you do not provide a value as well as for all columns that do not have default values associated with them, which are not listed explicitly. You also can use the NULL keyword to indicate that a column should be assigned a null value. The following example inserts values into three columns of the **orders** table:

```
INSERT INTO orders (orders_num, order_date, customer_num)
    VALUES (0, NULL, 123)
```

In this example, a null value is explicitly entered in the **order_date** column, and all other columns of the **orders** table that are *not* explicitly listed in the INSERT INTO clause are also filled with null values.

Subset of SELECT Statement

You can insert the rows of data that result from a SELECT statement into a table if the insert data is selected from another table or tables. If this statement has a WHERE clause that does not return rows, **sqlca** returns SQLNOTFOUND (100) for ANSI-compliant databases. In databases that are not ANSI compliant, **sqlca** returns (0). When you insert as a part of a multistatement prepare, and no rows are inserted, **sqlca** returns SQLNOTFOUND (100) for both ANSI databases and databases that are not ANSI compliant. The following SELECT clauses are not supported:

- INTO TEMP
- ORDER BY
- UNION

In addition, the FROM clause of the SELECT statement cannot contain the same table name as the table into which you are inserting rows, as shown in the following example:

```
INSERT INTO newtable
    SELECT item_num, order_num, quantity, stock_num,
           manu_code,  total_price
        FROM items
```

For detailed information on SELECT statement syntax, see page 1-459.

Using INSERT as a Dynamic Management Statement

ESQL

You can use the INSERT statement to handle situations where you need to write code that can insert data whose structure is unknown at the time you compile. For more information, refer to the dynamic management section of your SQL API manual. ♦

Inserting Data Using a Stored Procedure

You can insert the rows of data that result from a procedure call into a table.

The values that the procedure returns must match those expected by the column list in number and data type. The number and data types of the columns must match those that the column list expects.

References

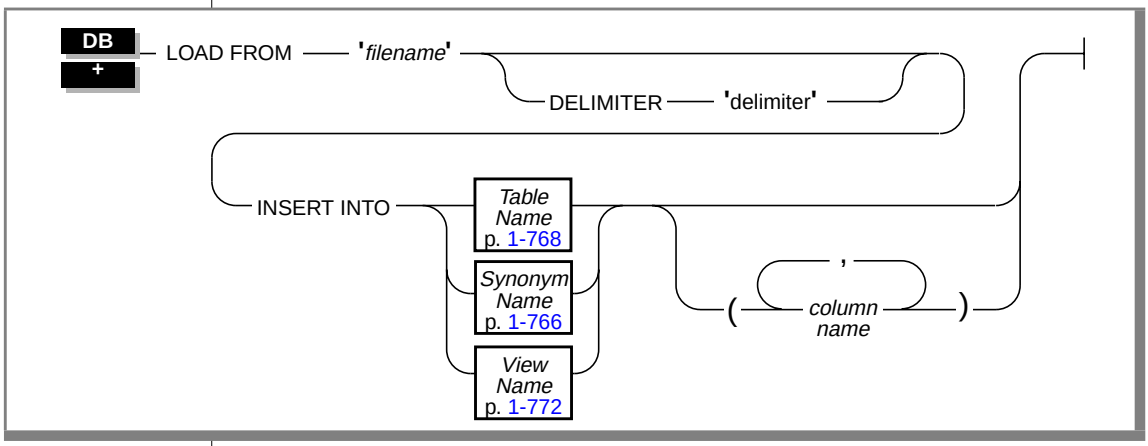
See the SELECT statement in this manual. Also see the DECLARE, DESCRIBE, EXECUTE, FLUSH, OPEN, PREPARE, and PUT statements in this manual for specific information about dynamic management statements.

In the *Informix Guide to SQL: Tutorial*, see the discussion of inserting data in Chapter 4 and Chapter 6. In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the INSERT statement.

LOAD

Use the LOAD statement to insert data from an operating-system file into an existing table, synonym, or view.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column or columns that receive data values from the load file during the load operation	You must specify the columns that receive data if you are not loading data into all columns. You must also specify columns if the order of the fields in the load file does not match the default order of the columns in the table (the order established when the table was created).	Identifier, p. 1-723
delimiter	A quoted string that identifies the character to use as the delimiter in the load file. The delimiter is a character that separates the data values in each line of the load file.	If you do not specify a delimiter character, the database server uses the setting in the **DBDELIMITER** environment variable. If **DBDELIMITER** has not been set, the default delimiter is the vertical bar (|). You cannot use the following items as delimiter characters: backslash (\ ), new-line character (=CTRL-J), and hexadecimal numbers (0-9, a-f, A-F).	Quoted String, p. 1-757
filename	A quoted string that identifies the pathname and filename of the load file. The load file contains the data to be loaded into the specified table or view. The default pathname for the load file is the current directory.	If you do not include a list of columns in the *column name* parameter, the fields in the load file must match the columns specified for the table in number, order, and type. You must also observe restrictions about the same number of fields in each line, the relationship of field lengths to column lengths, the representation of data types in the file, the use of the backslash character (\ ) with certain special characters, and special rules for VARCHAR and BLOB data types. See "The LOAD FROM File" on page 1-382 for information on these restrictions.	Quoted String, p. 1-757. The pathname and filename specified in the quoted string must conform to the conventions of your operating system.

Usage

The LOAD statement appends new rows to the table. It does not overwrite existing data.

You cannot add a row that has the same key as an existing row.

To use the LOAD statement, you must have Insert privileges for the table where you want to insert data. For information on database-level and table-level privileges, see the GRANT statement on page 1-340.

The LOAD FROM File

The LOAD FROM file contains the data to add to a table. You can use the file that the UNLOAD statement creates as the LOAD FROM file.

If you do not include a list of columns in the INSERT INTO clause, the fields in the file must match the columns that are specified for the table in number, order, and data type.

Each line of the file must have the same number of fields. You must define field lengths that are less than or equal to the length that is specified for the corresponding column. Specify only values that can convert to the data type of the corresponding column. The following table indicates how your Informix product expects you to represent the data types in the LOAD file (when they use the default locale, U.S. English).

Type of Data	Input Format
blank	One or more blank characters between delimiters. You can include leading blanks in fields that do not correspond to character columns.
date	A character string in the following format: *mm/dd/year*. You must state the month as a two-digit number. You can use a two-digit number for the year if the year is in the 20th century. (You can specify another century algorithm with the **DBCENTURY** environment variable.) The value must be an actual date; for example, February 30 is illegal. You can use a different date format if yo indicate this format with the **GL_DATE** or **DBDATE** environment variable. See Chapter 2 of the *Guide to GLS Functionality* for more information about these environment variables.
MONEY	A value that can include currency notation: a leading currency symbol ($), a comma (,) as the thousands separator, and a period (.) as the decimal separator. You can use different currency notation if you indicate this notation with the **DBMONEY** environment variable. For more information on this environment variable, see Chapter 2 of the *Guide to GLS Functionality*.
NULL	Nothing between the delimiters.
time	A character string in the following format: *year-month-day hour:minute:second.fraction*. You cannot use type specification or qualifiers for DATETIME or INTERVAL values. The year must be a four-digit number, and the month must be a two-digit number. You can specify a different date and time format with the **GL_DATETIME** or **DBTIME** environment variable. See Chapter 2 of the *Guide to GLS Functionality* for more information on these environment variables.

GLS

If you are using a nondefault locale, the formats of DATE, DATETIME, MONEY, and numeric column values in the LOAD FROM file must be compatible with the formats that the locale supports for these data types. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

If you include any of the following special characters as part of the value of a field, you must precede the character with a backslash (\):

- Backslash
- Delimiter
- New-line character anywhere in the value of a VARCHAR or NVARCHAR column
- New-line character at end of a value for a TEXT value

Do not use the backslash character (\) as a field separator. It serves as an escape character to inform the LOAD statement that the next character is to be interpreted as part of the data.

The fields that correspond to character columns can contain more characters than the defined maximum allows for the field. The extra characters are ignored.

If you are loading files that contain VARCHAR or BLOB data types, note the following information:

- If you give the LOAD statement data in which the character fields (including VARCHAR) are longer than the column size, the excess characters are disregarded.
- You cannot have leading and trailing blanks in BYTE fields.
- Use the backslash (\) to escape embedded delimiter and backslash characters in all character fields, including VARCHAR and TEXT.
- Data being loaded into a BYTE column must be in ASCII-hexadecimal form. BYTE columns cannot contain preceding blanks.
- Do not use the following as delimiting characters in the LOAD FROM file: 0 to 9, a to f, A to F, backslash, new-line character.

For more information about the format of the input file, see the discussion of the **dbload** utility in the *Informix Guide to SQL: Reference*.

The following example shows the contents of a hypothetical input file named **new_custs**:

```
0|Jeffery|Padgett|Wheel Thrills|3450 El Camino|Suite 10|Palo
Alto|CA|94306||
0|Linda|Lane|Palo Alto Bicycles|2344 University||Palo
Alto|CA|94301|(415)323-6440
```

This data file conveys the following information:

- Indicates a serial field by specifying a zero (0)
- Uses the vertical bar (|), the default delimiter character
- Assigns null values to the **phone** field for the first row and the **address2** field for the second row. The null values are shown by two delimiter characters with nothing between them.

The following statement loads the values from the **new_custs** file into the **customer** table owned by **jason**:

```
LOAD FROM 'new_custs' INSERT INTO jason.customer
```

DELIMITER Clause

Use the DELIMITER clause to specify the delimiter that separates the data contained in each column in a row in the input file. If you omit this clause, your Informix product checks the **DBDELIMITER** environment variable.

If the **DBDELIMITER** environment variable has not been set, the default delimiter is the vertical bar (|). See Chapter 4 in the *Informix Guide to SQL: Reference* for information about how to set the **DBDELIMITER** environment variable.

You can specify TAB (CTRL-I) or <blank> (= ASCII 32) as the delimiter symbol. You cannot use the following items as the delimiter symbol:

- Backslash (\)
- New-line character (= CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

The following statement identifies the semicolon (;) as the delimiting character:

```
LOAD FROM '/a/data/ord.loadfile' DELIMITER ';'
    INSERT INTO orders
```

INSERT INTO Clause

Use the INSERT INTO clause to specify the table, synonym, or view in which to load the new data. (See the discussion of Synonym Name, Table Name, and View Name that begins on page 1-766 for details.)

You must specify the column names only if one of the following conditions is true:

- You are not loading data into all columns.
- The input file does not match the default order of the columns (determined when the table was created).

The following example identifies the **price** and **discount** columns as the only columns in which to add data:

```
LOAD FROM '/tmp/prices' DELIMITER ','
    INSERT INTO norman.worktab(price,discount)
```

References

See the UNLOAD and INSERT statements in this manual.

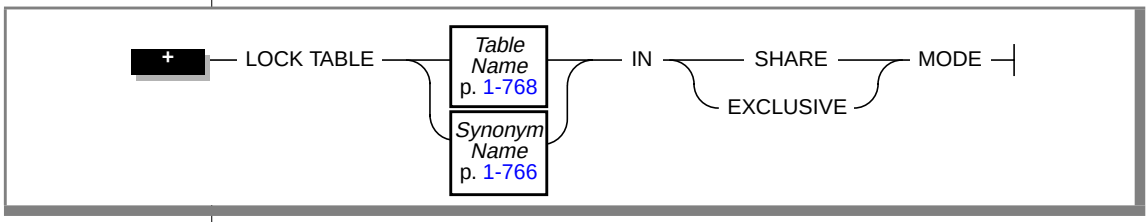
In the *Informix Migration Guide*, see the task-oriented discussion of the LOAD statement and other utilities for moving data.

In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the LOAD statement.

LOCK TABLE

Use the LOCK TABLE statement to control access to a table by other processes.

Syntax



Usage

You can lock a table if you own the table or have the Select privilege on the table or on a column in the table, either from a direct grant or from a grant to PUBLIC. The LOCK TABLE statement fails if the table is already locked in exclusive mode by another process, or if an exclusive lock is attempted while another user has locked the table in share mode.

The SHARE keyword locks a table in shared mode. Shared mode allows other processes *read* access to the table but denies *write* access. Other processes cannot update or delete data if a table is locked in shared mode.

The EXCLUSIVE keyword locks a table in exclusive mode. Exclusive mode denies other processes both *read* and *write* access to the table.

Exclusive-mode locking automatically occurs when you execute the ALTER INDEX, CREATE INDEX, DROP INDEX, RENAME COLUMN, RENAME TABLE, and ALTER TABLE statements.

SE

The INFORMIX-SE database server does not permit more than one user to lock a table in shared mode. ♦

Databases with Transactions

If your database was created with transactions, the LOCK TABLE statement succeeds only if it executes within a transaction. You must issue a BEGIN WORK statement before you can execute a LOCK TABLE statement.

ANSI

Transactions are implicit in an ANSI-compliant database. The LOCK TABLE statement succeeds whenever the specified table is not already locked by another process. ♦

The following guidelines apply to the use of the LOCK TABLE statement within transactions:

- You cannot lock system catalog tables.
- You cannot switch between shared and exclusive table locking within a transaction. For example, once you lock the table in shared mode, you cannot upgrade the lock mode to exclusive.
- If you issue a LOCK TABLE statement before you access a row in the table, no row locks are set for the table. In this way, you can override row-level locking and avoid exceeding the maximum number of locks that are defined in the OnLine configuration.

  SE

  The maximum number of locks that are allowed by the INFORMIX-SE database server is a characteristic of the particular operating system on which your database server is running. ♦
- All row and table locks release automatically after a transaction is completed. Note that the UNLOCK TABLE statement fails within a database that uses transactions.

The following example shows how to change the locking mode of a table in a database that was created with transaction logging:

```
BEGIN WORK
LOCK TABLE orders IN EXCLUSIVE MODE
 ...
COMMIT WORK
BEGIN WORK
LOCK TABLE orders IN SHARE MODE
 ...
COMMIT WORK
```

Databases Without Transactions

In a database that was created without transactions, table locks set by using the LOCK TABLE statement are released after any of the following occurrences:

- An UNLOCK TABLE statement executes.
- The user closes the database.
- The user exits the application.

To change the lock mode on a table, release the lock with the UNLOCK TABLE statement and then issue a new LOCK TABLE statement.

The following example shows how to change the lock mode of a table in a database that was created without transactions:

```
LOCK TABLE orders IN EXCLUSIVE MODE
  .
  .
  .
UNLOCK TABLE orders
  .
  .
  .
LOCK TABLE orders IN SHARE MODE
```

References

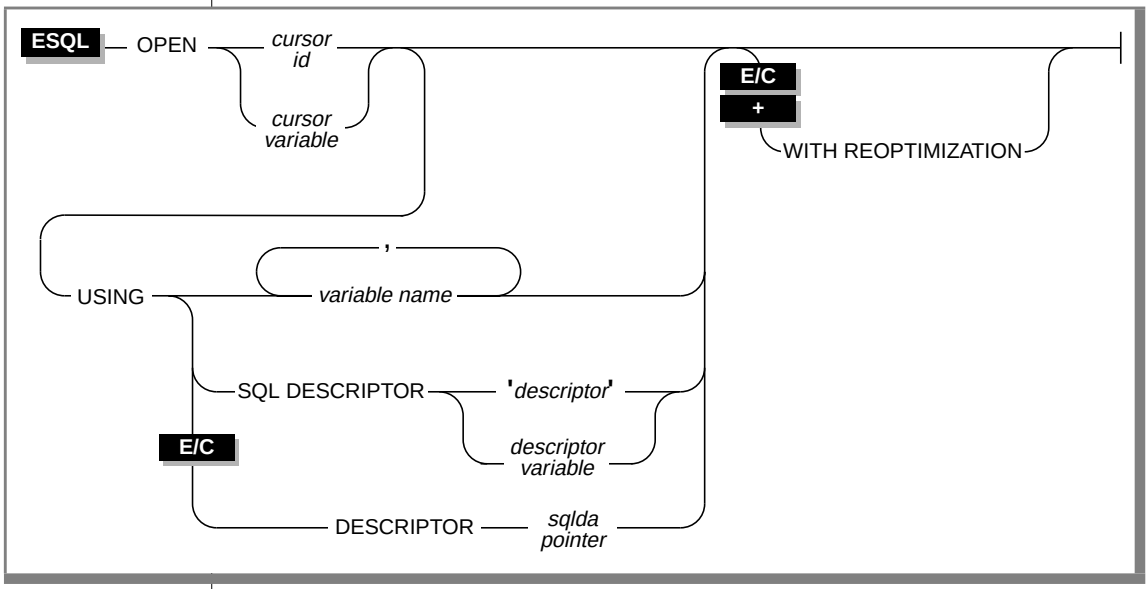
See the BEGIN WORK, SET ISOLATION, SET LOCK MODE, COMMIT WORK, ROLLBACK WORK, and UNLOCK TABLE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of locks in Chapter 7.

OPEN

Use the OPEN statement to activate a cursor associated with a SELECT, INSERT, or EXECUTE PROCEDURE statement, and thereby begin execution of the SELECT, INSERT, or EXECUTE PROCEDURE statement.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	It identifies a cursor	Cursor must have been previously created by a DECLARE statement.	Identifier, p. 1-723
cursor variable	Host variable that identifies a cursor	Host variable must be a character data type. Cursor must have been previously created by a DECLARE statement.	Variable name must conform to language-specific rules for variable names
descriptor	Quoted string that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-757

(1 of 2)

Element	Purpose	Restrictions	Syntax
descriptor variable	Host variable name that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-757
sqlda pointer	It points to an **sqlda** structure that defines the type and memory location of values that correspond to the question-mark (?) placeholder in a prepared statement	You cannot begin an *sqlda pointer* with a dollar sign ($) or a colon (:). You must use an **sqlda** structure if you are using dynamic SQL statements.	DESCRIBE, p. 1-255
variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

Usage

You create a cursor with a statement that uses the DECLARE statement (see page 1-234). When the program opens the cursor, the associated SELECT, INSERT, or EXECUTE PROCEDURE statement is passed to the database server, which begins execution. When the program has retrieved or inserted all the rows it needs, close the cursor by using the CLOSE statement.

The specific actions that the database server takes differ, depending on whether the cursor is associated with a SELECT statement or an INSERT statement.

The SELECT, INSERT, or EXECUTE PROCEDURE statement associated with a cursor is prepared implicitly by the OPEN statement. The total number of prepared objects and open cursors that are allowed in one program at any time is limited by the available memory. You can use the FREE statement to free the cursor and release the database server resources.

ANSI

You receive an error code if you try to open a cursor that is already open. ♦

Opening a Select Cursor

When you open either a select cursor or an update cursor, the SELECT statement is passed to the database server along with any values that are specified in the USING clause. (If the statement was previously prepared, the statement passed to the database server when it was prepared.) The database server processes the query to the point of locating or constructing the first row of the active set.

Because the database server is seeing the query for the first time, many errors are detected. The database server does not actually return the first row of data, but it sets a return code in the **SQLCODE** field of the **sqlca**. The name of the field in each <vk>SQL API product is indicated in the following table.

Product	Field Name
ESQL / C	sqlca.sqlcode, SQLCODE
ESQL / COBOL	SQLCODE OF SQLCA

The return code value is either negative or zero, as the following table describes.

Return Code Value	Meaning
Negative	Shows an error is detected in the SELECT statement.
Zero	Shows the SELECT statement is valid.

If the SELECT statement is valid, but no rows match its criteria, the first FETCH statement returns a value of 100 (SQLNOTFOUND), which means no rows were found.

Tip: *When you encounter an **SQLCODE** error, a corresponding **SQLSTATE** error value might exist. Check the GET DIAGNOSTICS statement for information about how to get the **SQLSTATE** value and how to use the GET DIAGNOSTICS statement to interpret the **SQLSTATE** value.*

The following example illustrates a simple OPEN statement in INFORMIX-ESQL / C:

```
EXEC SQL declare s_curs cursor for
    select * from orders;
EXEC SQL open s_curs;
```

If you are working in a database with explicit transactions, you must open an update cursor within a transaction. This requirement is waived if you declared the cursor using the WITH HOLD keyword. (See the DECLARE statement on page 1-234.)

Opening a Procedure Cursor

When you open a procedure cursor, the EXECUTE PROCEDURE statement is passed to the database server along with any values that are specified in the USING clause. The values are passed as arguments to the stored procedure, and the procedure must be declared to accept values. (If the statement was previously prepared, the statement passed to the database server when it was prepared.) The database server executes the procedure to the point of the first set of values returned by the procedure.

Because the database server is seeing the procedure for the first time, many errors are detected. The database server does not actually return the first row of data, but it sets a return code in the **SQLCODE** field of the **sqlca**. The name of the field in each product is indicated in the following table.

Product	Field Name
ESQL / C	sqlca.sqlcode, SQLCODE
ESQL / COBOL	SQLCODE OF SQLCA

The return-code value is either negative or zero, as described in the following table.

Return Code Value	Meaning
Negative	Shows that an error was detected in the EXECUTE PROCEDURE statement.
Zero	Shows that the EXECUTE PROCEDURE statement is valid.

If the EXECUTE PROCEDURE statement is valid, but no rows are returned, the first FETCH statement returns a value of 100 (SQLNOTFOUND), which means no rows found. The procedure must be created to return values; that is, the procedure must have a RETURNING clause at the beginning of the procedure.

Tip: *When you encounter an **SQLCODE** error, be aware that there may be a corresponding **SQLSTATE** error value. Check the GET DIAGNOSTICS statement for information about how to get the **SQLSTATE** value and how to use the GET DIAGNOSTICS statement to interpret the **SQLSTATE** value.*

The following example illustrates a simple OPEN statement in INFORMIX-ESQL / C:

```
EXEC SQL declare s_curs cursor for
    execute procedure new_proc();
EXEC SQL open s_curs;
```

Opening an Insert Cursor

When you open an insert cursor, the cursor passes the INSERT statement to the database server, which checks the validity of the keywords and column names. The database server also allocates memory for an insert buffer to hold new data. (See the DECLARE statement on page 1-234.)

An OPEN statement for a cursor that is associated with an INSERT statement cannot include a USING clause.

The following INFORMIX-ESQL/C example illustrates an OPEN statement with an insert cursor:

```
EXEC SQL prepare s1 from
    'insert into manufact values ('npr', 'napier')';
EXEC SQL declare in_curs cursor for s1;
EXEC SQL open in_curs;
EXEC SQL put in_curs;
EXEC SQL close in_curs;
```

Reopening a Select Cursor

The values that are named in the USING clause are evaluated only when the cursor is opened. While the cursor is open, subsequent changes to program variables in the USING clause do not change the active set of selected rows. The active set remains constant until a subsequent OPEN statement closes the cursor and reopens it or until the program closes the open cursor, which releases the active set.

Reopening the cursor creates a new active set that is based on the current values of the variables. If the program variables have changed since the previous OPEN statement, reopening the cursor can generate an entirely different active set. Even if the values of the variables are unchanged, if data in the table was modified since the previous OPEN statement, the rows in the active set can be different.

Reopening a Procedure Cursor

The values that are named in the USING clause are evaluated only when the cursor is opened. While the cursor is open, subsequent changes to program variables in the USING clause do not change the active set of returned rows. The active set remains constant until a subsequent OPEN statement closes the cursor and reopens it or until the program closes the open cursor, which releases the active set.

Reopening the cursor creates a new active set that is based on the current values of the variables. If the program variables have changed since the previous OPEN statement, reopening the cursor can generate an entirely different active set. Even if the values of the variables are unchanged, if the procedure takes a different execution path from the previous OPEN statement, the rows in the active set can be different.

Reopening an Insert Cursor

When you reopen an insert cursor that is already open, you effectively flush the insert buffer; any rows that are stored in the insert buffer are written into the database table. The database server first closes the cursor, which causes the flush and then reopens the cursor. See the discussion of the PUT statement on page 1-416 for information about checking errors and counting inserted rows.

USING Clause

The USING clause is required when the cursor is associated with a prepared SELECT statement that includes question-mark (?) placeholders. (See the PREPARE statement on page 1-402.) You can supply values for these parameters in one of two ways. You can specify host variables in the USING clause, or you can specify a system-descriptor area in the USING SQL DESCRIPTOR clause.

Naming Variables in USING

If you know the number of parameters to be supplied at runtime and their data types, you can define the parameters that are needed by the statement as host variables in your program. You pass parameters to the database server by opening the cursor with the USING keyword, followed by the names of the variables. These variables are matched with the SELECT statement question-mark (?) parameters in a one-to-one correspondence, from left to right.

You cannot include indicator variables in the list of variable names. To use an indicator variable, you must include the SELECT statement as part of the DECLARE statement.

The following example illustrates the USING clause with the OPEN statement in an INFORMIX-ESQL / C code fragment:

```
sprintf (select_1, "%s %s %s %s %s",
    "SELECT o.order_num, sum(total price)",
    "FROM orders o, items i",
    "WHERE o.order_date > ? AND o.customer_num = ?",
    "AND o.order_num = i.order_num",
    "GROUP BY o.order_num");
EXEC SQL prepare statement_1 from :select_1;
EXEC SQL declare q_curs cursor for statement_1;
EXEC SQL open q_curs using :o_date, :o_total;
```

USING SQL DESCRIPTOR Clause

You can also associate input values from a system-descriptor area. The keywords USING SQL DESCRIPTOR indicate the use of a system descriptor. This allows you to associate input values from a system-descriptor area and open a cursor.

If a system-descriptor area is used, the **count** value specifies the number of input values that are described in occurrences of **sqlvar**. This number must correspond to the number of dynamic parameters in the prepared statement. The value of **count** must be less than or equal to the value of occurrences that were specified when the system-descriptor area was allocated.

For further information, refer to the discussion of the system-descriptor area in the manual for your SQL API. The following examples show the OPEN ... USING SQL DESCRIPTOR clause:

INFORMIX-ESQL/C

```
EXEC SQL open selcurs using sql descriptor 'desc1';
```

INFORMIX-ESQL/COBOL

```
EXEC SQL OPEN SEL_CURS USING SQL DESCRIPTOR 'DESC1' END-EXEC.
```

USING DESCRIPTOR Clause

E/C

You can pass parameters for a prepared statement in the form of an **sqlda** pointer structure, which lists the data type and memory location of one or more values to replace question-mark (?) placeholders. For further information, refer to the **sqlda** discussion in the *INFORMIX-ESQL/C Programmer's Manual*. The following example shows the OPEN ... USING DESCRIPTOR clause in INFORMIX-ESQL / C:

```
struct sqlda *sdp;
.
.
.
EXEC SQL open selcurs using descriptor sdp;
```

◆

WITH REOPTIMIZATION Clause

The WITH REOPTIMIZATION clause allows you to reoptimize your query-design plan. When you prepare a SELECT statement or an EXECUTE PROCEDURE statement, your Informix database server uses a query-design plan to optimize that query. If you later modify the data that is associated with a prepared SELECT statement or the data that is associated with an EXECUTE PROCEDURE statement, you can compromise the effectiveness of the query-design plan for that statement. In other words, if you change the data, you can deoptimize your query. To ensure optimization of your query, you can prepare the SELECT or EXECUTE PROCEDURE statement again or open the cursor again using the WITH REOPTIMIZATION clause. Informix recommends that you use the WITH REOPTIMIZATION clause because it provides the following advantages over preparing a statement again:

- Rebuilds only the query-design plan rather than the entire statement
- Uses fewer resources
- Reduces overhead
- Requires less time

The WITH REOPTIMIZATION clause also makes your database server optimize your query-design plan before processing the OPEN cursor statement.

The following example shows the WITH REOPTIMIZATION clause in INFORMIX-ESQL / C and INFORMIX-ESQL / COBOL, respectively:

INFORMIX-ESQL/C

```
EXEC SQL open selcurs using descriptor sdp with reoptimization;
```

INFORMIX-ESQL/COBOL

```
EXEC SQL OPEN SEL_CURS USING SQL DESCRIPTOR 'DESC1' WITH REOPTIMIZATION END-EXEC.
```

The Relationship Between OPEN and FREE

The database server allocates resources to prepared statements and open cursors. If you release resources with a FREE *cursor id* or FREE *cursor variable* statement, you cannot use the cursor unless you declare the cursor again. If you execute a FREE *statement id* or FREE *statement id variable* statement, you cannot open the cursor that is associated with the statement id or statement id variable unless you prepare the statement id or statement id variable again.

References

See the CLOSE, DECLARE, and FREE statements in this manual for cursor-related information. See the PUT and FLUSH statements in this manual for information about insert cursors.

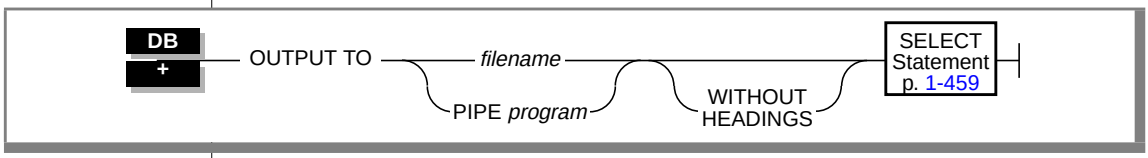
See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, PREPARE, PUT, and SET DESCRIPTOR statements in this manual for more information about dynamic <vk>SQL statements.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the OPEN statement in Chapter 5. Refer also to your SQL API manual for more information about the system-descriptor area and the **sqlda** structure.

OUTPUT

Use the OUTPUT statement to send query results directly to an operating-system file or to pipe it to another program.

Syntax



Element	Purpose	Restrictions	Syntax
filename	The pathname and filename of an operating-system file where the results of the query are written. The default pathname is the current directory.	You can specify a new or existing file in *filename*. If the specified file exists, the results of the query overwrite the current contents of the file.	The pathname and filename must conform to the conventions of your operating system.
program	The name of a program where the results of the query are sent	The program must exist and must be known to the operating system. The program must be able to read the results of a query.	The name of the program must conform to the conventions of your operating system.

Usage

You can send the results of a query to an operating-system file by specifying the full pathname for the file. If the file already exists, the output overwrites the current contents, as the following example shows:

```
OUTPUT TO /usr/april/query1
    SELECT * FROM cust_calls WHERE call_code = 'L'
```

You can display the results of a query without column headings by using the WITHOUT HEADINGS keywords, as the following example shows:

```
OUTPUT TO /usr/april/query1
    WITHOUT HEADINGS
    SELECT * FROM cust_calls WHERE call_code = 'L'
```

You also can use the keyword PIPE to send the query results to another program, as the following example shows:

```
OUTPUT TO PIPE more
    SELECT customer_num, call_dtime, call_code
        FROM cust_calls
```

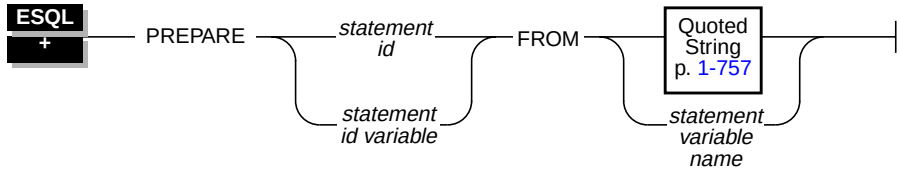
References

See the SELECT and UNLOAD statements in this manual.

PREPARE

Use the PREPARE statement to parse, validate, and generate an execution plan for SQL statements in an SQL API program at runtime.

Syntax



Element	Purpose	Restrictions	Syntax
statement id	The statement identifier identifies an SQL statement.	After you release the database-server resources (using a FREE statement), you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.	Identifier, p. 1-723
statement id variable	Host variable that contains the statement identifier	This variable must be a character data type.	Variable name must conform to language-specific rules for variable names.
statement variable name	Host variable whose value is a character string that consists of one or more SQL statements	This variable must be a character data type. For restrictions on the statements in the character string, see "SQL Statements Permitted in Single-Statement Prepares" on page 1-407 and "Restrictions for Multistatement Prepares" on page 1-414.	Variable name must conform to language-specific rules for variable names.

Usage

The PREPARE statement permits your program to assemble the text of an SQL statement at runtime and make it executable. This dynamic form of SQL is accomplished in three steps:

1. A PREPARE statement accepts statement text as input, either as a quoted string or stored within a character variable. Statement text can contain question-mark (?) placeholders to represent values that are to be defined when the statement is executed.
2. An EXECUTE or OPEN statement can supply the required input values and execute the prepared statement once or many times.
3. Resources allocated to the prepared statement can be released later using the FREE statement.

The number of prepared objects in a single program is limited by the available memory. This includes both statement identifiers that are named in PREPARE statements (*statement id* or *statement id variable*) and cursor declarations that incorporate SELECT, EXECUTE PROCEDURE, or INSERT statements. To avoid exceeding the limit, use a FREE statement to release some statements or cursors.

Using a Statement Identifier

A PREPARE statement sends the statement text to the database server where it is analyzed. If it contains no syntax errors, the text converts to an internal form. This translated statement is saved for later execution in a data structure that the PREPARE statement allocates. The name of the structure is the value that is assigned to the statement identifier in the PREPARE statement. Subsequent SQL statements refer to the structure by using the same statement identifier that was used in the PREPARE statement.

A subsequent FREE statement releases the resources that were allocated to the statement. After you release the database-server resources, you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.

A program can consist of one or more source-code files. By default, the scope of a statement identifier is global to the program. Therefore, a statement identifier that is prepared in one file can be referenced from another file.

In a multiple-file program, if you want to limit the scope of a statement identifier to the file in which it is prepared, preprocess all the files with the **-local** command-line option. See the manual for your SQL API for more information, restrictions, and performance issues when preprocessing with the **-local** option.

Releasing a Statement Identifier

A statement identifier can represent only one SQL statement or sequence of statements at a time. You can execute a new PREPARE statement with an existing statement identifier if you wish to bind a given statement identifier to a different SQL statement text.

The PREPARE statement supports dynamic statement-identifier names, which allow you to prepare a statement identifier as an identifier or as a host character-string variable. In the following ESQL/C and ESQL/COBOL examples, the first example in each pair shows a statement identifier that was prepared as an SQL API variable; the second example in each pair shows a statement identifier that was prepared as a character-string constant:

INFORMIX-ESQL/C

```
stcopy ("query2", stmtid);
EXEC SQL prepare :stmtid from
    'select * from customer';

EXEC SQL prepare query2 from
    'select * from customer';
```

INFORMIX-ESQL/COBOL

```
MOVE 'QUERY_2' TO STMTID.
EXEC SQL
 PREPARE :STMTID FROM
    'SELECT * FROM CUSTOMER'
END-EXEC.

EXEC SQL
 PREPARE QUERY_2 FROM
    'SELECT * FROM CUSTOMER'
END-EXEC.
```

A statement ID variable must be the CHARACTER data type. In C, it must be defined as `char`. In COBOL, ID variables must be declared as a standard CHARACTER type.

Statement Text

The PREPARE statement can take statement text either as a quoted string or as text that is stored in a program variable. The following restrictions apply to the statement text:

- The text can contain only SQL statements. It cannot contain statements or comments *from* the host programming language.
- The text can contain comments that are preceded by a double dash (--) or enclosed in curly brackets ({ }). These comment symbols represent SQL comments. For more information on SQL comment symbols, see "How to Enter SQL Comments" on page 1-9.
- The text can contain either a single SQL statement or a sequence of statements that are separated by semicolons. For more information on preparing a single SQL statement, see "SQL Statements Permitted in Single-Statement Prepares" on page 1-407. For more information on preparing a sequence of SQL statements, see "Preparing Sequences of Multiple SQL Statements" on page 1-412.
- Names of host-language variables are not recognized as such in prepared text. Therefore, you cannot prepare a SELECT statement that contains an INTO clause or an EXECUTE PROCEDURE that contains an INTO clause because the INTO clause requires a host-language variable.
- The only identifiers that you can use are names that are defined in the database, such as names of tables and columns. For further information on using identifiers in statement text, see "Preparing Statements with SQL Identifiers" on page 1-409.
- Use a question mark (?) as a placeholder to indicate where data is supplied when the statement executes. For further information on using question marks as placeholders, see "Preparing Statements That Receive Parameters" on page 1-408.
- The text cannot include an embedded SQL statement prefix or terminator, such as a dollar sign ($) or the words EXEC SQL.

The following example shows a PREPARE statement in INFORMIX-ESQL/C:

```
EXEC SQL prepare new_cust from
    'insert into customer(fname,lname) values(?,?)';
```

Executing Stored Procedures Within a PREPARE Statement

You can prepare an EXECUTE PROCEDURE statement as long as it does not contain an INTO clause. The way to execute a prepared stored procedure depends on whether the stored procedure returns values:

- If the stored procedure does not return values (the procedure does not contain the RETURN statement), use the EXECUTE statement to execute the EXECUTE PROCEDURE statement.
- If the stored procedure returns only one row, you can use the INTO clause of the EXECUTE statement to specify host variables to hold the return values. Using EXECUTE ... INTO to execute a stored procedure that returns more than one row generates a runtime error.
- If the stored procedure returns more than one row, you must associate the prepared EXECUTE PROCEDURE statement with a cursor using the DECLARE statement. You execute the statement with the OPEN statement and retrieve return values into host variables using the INTO clause of the FETCH statement.

If you do not know the number and data types of the values that a stored procedure returns, you must use a dynamic management structure to hold the returned values. In an ESQL/C or ESQL/COBOL program, you can use SQL statements such as ALLOCATE DESCRIPTOR and GET DESCRIPTOR to manage a system-descriptor area. In an ESQL/C program, you can use an **sqlda** structure instead of a system-descriptor area. However, a system-descriptor area conforms to the X/Open standards.

See Chapter 12 of the *Informix Guide to SQL: Tutorial* for information about creating and executing stored procedures. See the *INFORMIX-ESQL/COBOL Programmer's Manual* and the *INFORMIX-ESQL/C Programmer's Manual* for detailed information about using dynamic management structures to dynamically execute a stored procedure.

SQL Statements Permitted in Single-Statement Prepares

You can prepare any single SQL statement except the ones in the following list.

ALLOCATE DESCRIPTOR	GET DIAGNOSTICS
CHECK TABLE	INFO
CLOSE	LOAD
DEALLOCATE DESCRIPTOR	OPEN
DECLARE	OUTPUT
DESCRIBE	PREPARE
EXECUTE IMMEDIATE	PUT
EXECUTE	REPAIR TABLE
FETCH	SET DESCRIPTOR
FLUSH	UNLOAD
FREE	WHENEVER
GET DESCRIPTOR	

You can prepare a SELECT statement. If the SELECT statement includes the INTO TEMP clause, you can execute the prepared statement with an EXECUTE statement. If it does not include the INTO TEMP clause, the statement returns rows of data. Use DECLARE, OPEN, and FETCH cursor statements to retrieve the rows.

A prepared SELECT statement can include a FOR UPDATE clause. This clause is normally used with the DECLARE statement to create an update cursor. The following example shows a SELECT statement with a FOR UPDATE clause in INFORMIX-ESQL / C:

```
sprintf(up_query, "%s %s %s",
    "select * from customer ",
    "where customer_num between ? and ? ",
    "for update");
EXEC SQL prepare up_sel from :up_query;

EXEC SQL declare up_curs cursor for up_sel;

EXEC SQL open up_curs using :low_cust,:high_cust;
```

Preparing Statements When Parameters Are Known

In some prepared statements, all needed information is known at the time the statement is prepared. The following example in INFORMIX-ESQL / C shows two statements that were prepared from constant data:

```
sprintf(redo_st, "%s %s",
    "drop table workt1; ",
    "create table workt1 (wtk serial, wtv float)" );
EXEC SQL prepare redotab from :redo_st;
```

Preparing Statements That Receive Parameters

In some statements, parameters are unknown when the statement is prepared because a different value can be inserted each time the statement is executed. In these statements, you can use a question-mark (?) placeholder where a parameter must be supplied when the statement is executed.

The PREPARE statements in the following INFORMIX-ESQL / C examples show some uses of question-mark (?) placeholders:

```
EXEC SQL prepare s3 from
    'select * from customer where state matches ?';

EXEC SQL prepare in1 from
    'insert into manufact values (?,?,?)';

sprintf(up_query, "%s %s",
    "update customer set zipcode = ?"
    "where current of zip_cursor");
EXEC SQL prepare update2 from :up_query;
```

You can use a placeholder to defer evaluation of a value until runtime only for an expression. You cannot use a question-mark (?) placeholder to represent an SQL identifier except as noted in "Preparing Statements with SQL Identifiers" on page 1-409.

The following example of an INFORMIX-ESQL/C code fragment prepares a statement from a variable that is named **demoquery**. The text in the variable includes one question-mark (?) placeholder. The prepared statement is associated with a cursor and, when the cursor is opened, the USING clause of the OPEN statement supplies a value for the placeholder.

```
EXEC SQL BEGIN DECLARE SECTION;
    char queryvalue [6];
    char demoquery  [80];
EXEC SQL END DECLARE SECTION;

EXEC SQL connect to 'stores7';
sprintf(demoquery, "%s %s",
        "select fname, lname from customer ",
        "where lname > ? ");
EXEC SQL prepare quid from :demoquery;
EXEC SQL declare democursor cursor for quid;
stcopy("C", queryvalue);
EXEC SQL open democursor using :queryvalue;
```

The USING clause is available in both OPEN (for statements that are associated with a cursor) and EXECUTE (all other prepared statements) statements.

Preparing Statements with SQL Identifiers

In general, you cannot use question-mark (?) placeholders for SQL identifiers. You must specify these identifiers in the statement text when you prepare the statement.

However, in a few special cases, you can use the question mark (?) placeholder for an SQL identifier. These cases are as follows:

- You can use the *?* placeholder for the database name in the DATABASE statement.
- You can use the *?* placeholder for the dbspace name in the IN *dbspace* clause of the CREATE DATABASE statement
- You can use the *?* placeholder for the cursor name in statements that use cursor names. ♦

E/C

Obtaining SQL Identifiers from User Input

If a prepared statement requires identifiers, but the identifiers are unknown when you write the prepared statement, you can construct a statement that receives SQL identifiers from user input.

The following INFORMIX-ESQL / C example prompts the user for the name of a table and uses that name in a SELECT statement. Because the table name is unknown until runtime, the number and data types of the table columns are also unknown. Therefore, the program cannot allocate host variables to receive data from each row in advance. Instead, this program fragment describes the statement into an **sqlda** descriptor and fetches each row using the descriptor. The fetch puts each row into memory locations that the program provides dynamically.

If a program retrieves all the rows in the active set, the FETCH statement would be placed in a loop that fetched each row. If the FETCH statement retrieves more than one data value (column), another loop exists after the FETCH, which performs some action on each data value.

```
#include <stdio.h>
EXEC SQL include sqlda;
EXEC SQL include sqltypes;

char *malloc( );

main()
{
    struct sqlda *demodesc;
    char tablename[19];
    int i;
EXEC SQL BEGIN DECLARE SECTION;
    char demoselect[200];
EXEC SQL END DECLARE SECTION;

/*  This program selects all the columns of a given tablename.
       The tablename is supplied interactively. */

EXEC SQL connect to 'stores7';

printf( "This program does a select * on a table\n" );
printf( "Enter table name: " );
scanf( "%s", tablename );

sprintf(demoselect, "select * from %s", tablename );

EXEC SQL prepare iid from :demoselect;
EXEC SQL describe iid into demodesc;

/* Print what describe returns */

for ( i = 0;  i < demodesc->sqld; i++ )
    prsqlda (demodesc->sqlvar + i);
```

```
/* Assign the data pointers. */

for ( i = 0;  i < demodesc->sqld; i++ )
    {
    switch (demodesc->sqlvar[i].sqltype & SQLTYPE)
        {
        case SQLCHAR:
            demodesc->sqlvar[i].sqltype = CCHARTYPE;
            demodesc->sqlvar[i].sqllen++;
            demodesc->sqlvar[i].sqldata =
                malloc( demodesc->sqlvar[i].sqllen );
            break;

        case SQLSMINT:    /* fall through */
        case SQLINT:      /* fall through */
        case SQLSERIAL:
            demodesc->sqlvar[i].sqltype = CINTTYPE;
            demodesc->sqlvar[i].sqldata =
                malloc( sizeof( int ) );
            break;

        /*  And so on for each type.  */

        }
    }

/* Declare and open cursor for select . */
EXEC SQL declare d_curs cursor for iid;
EXEC SQL open d_curs;

/* Fetch selected rows one at a time into demodesc. */

for( ; ; )
    {
    printf( "\n" );
    EXEC SQL fetch d_curs using descriptor demodesc;
    if ( sqlca.sqlcode != 0 )
          break;
    for ( i = 0;  i < demodesc->sqld; i++ )
        {
        switch (demodesc->sqlvar[i].sqltype)
            {
            case CCHARTYPE:
                printf( "%s: \"%s\n", demodesc->sqlvar[i].sqlname,
                    demodesc->sqlvar[i].sqldata );
                break;
            case CINTTYPE:
                printf( "%s: %d\n", demodesc->sqlvar[i].sqlname,
                    *((int *) demodesc->sqlvar[i].sqldata) );
                break;

            /* And so forth for each type... */

            }
        }
    }
EXEC SQL close d_curs;
EXEC SQL free d_curs;
```

```
/*  Free the data memory.  */

for ( i = 0;  i < demodesc->sqld; i++ )
    free( demodesc->sqlvar[i].sqldata );

printf ("Program Over.\n");
}


prsqlda(sp)
    struct sqlvar_struct *sp;
    {
    printf ("type = %d\n", sp->sqltype);
    printf ("len = %d\n", sp->sqllen);
    printf ("data = %lx\n", sp->sqldata);
    printf ("ind = %lx\n", sp->sqlind);
    printf ("name = %s\n", sp->sqlname);
    }
```

For an explanation of how to use an **sqlda** structure for statement values, see the *INFORMIX-ESQL/C Programmer's Manual*.

Preparing Sequences of Multiple SQL Statements

You can execute several SQL statements as one action if you include them in the same PREPARE statement. Multistatement text is processed as a unit; actions are not treated sequentially. Therefore, multistatement text cannot include statements that depend on actions that occur in a previous statement in the text. For example, you cannot create a table and insert values into that table in the same prepared block.

In most situations, compiled products return error-status information on the first error in the multistatement text. No indication exists of which statement in the sequence causes an error. You can use **sqlca** to find the offset of the following errors:

- In ESQL / C: **sqlca.sqlerrd[4]**
- In ESQL / COBOL: SQLERRD[5] OF SQLCA

For more information about **sqlca** and error-status information, see your SQL API manual.

In a multistatement prepare, if no rows are returned from a WHERE clause in the following statements, you get SQLNOTFOUND (100) in both ANSI-compliant databases and databases that are not ANSI compliant:

- UPDATE ... WHERE ...
- SELECT INTO TEMP ... WHERE ...
- INSERT INTO ... WHERE ...
- DELETE FROM ...WHERE ...

In the following example, four SQL statements are prepared into a single INFORMIX-ESQL/C string that is called **query**. Individual statements are delimited with semicolons. A single PREPARE statement can prepare the four statements for execution, and a single EXECUTE statement can execute the statements that are associated with the **qid** statement identifier.

```
sprintf (query,  "%s %s %s %s %s %s %s",
    "update account set balance = balance + ? ",
        "where acct_number = ?;",
    "update teller set balance = balance + ? ",
        "where teller_number = ?;",
    "update branch set balance = balance + ? ",
        "where branch_number = ?;",
    "insert into history values (?, ?);";
EXEC SQL prepare qid from :query;

EXEC SQL begin work;
EXEC SQL execute qid using
        :delta, :acct_number, :delta, :teller_number,
        :delta, :branch_number, :timestamp, :values;
EXEC SQL commit work;
```

In the preceding code fragment, the semicolons (;) are required as SQL statement-terminator symbols between each SQL statement in the text that **query** holds.

Restrictions for Multistatement Prepares

In addition to the statements listed in "SQL Statements Permitted in Single-Statement Prepares" on page 1-407, you cannot use the following statements

```
            printf("Enter customer number (or 0 to quit): ");
            scanf("%d", cust_num);
            if (cust_num == 0)
                break;
            EXEC SQL execute up1 using :cust_num;
            }
    }
```

References

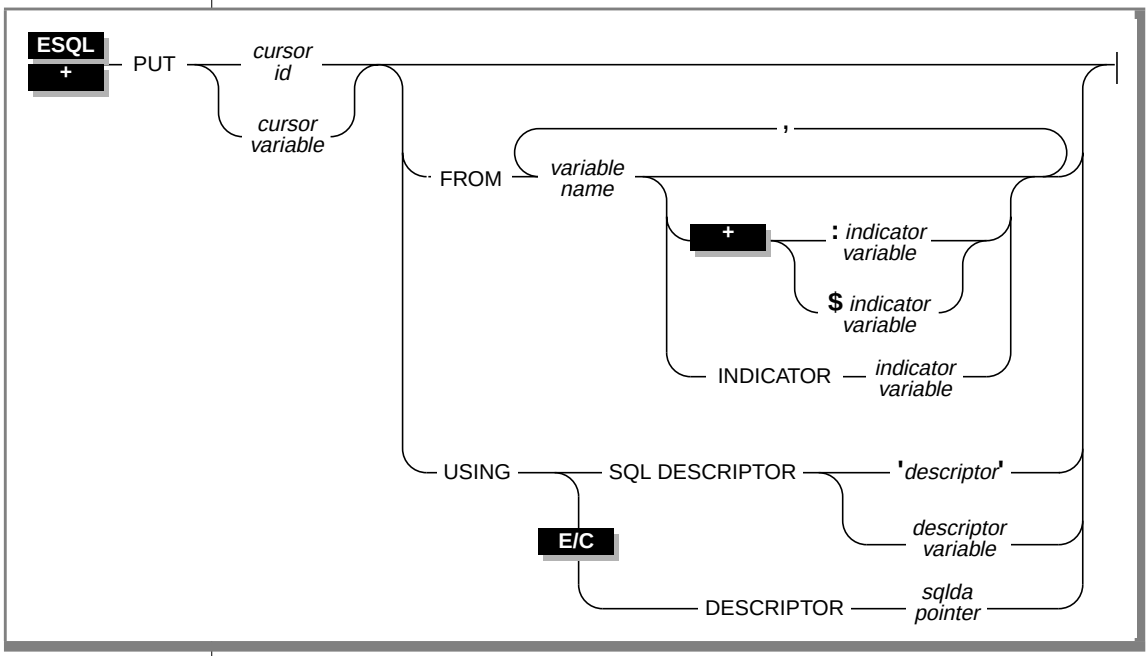
See the DECLARE, DESCRIBE, EXECUTE, FREE, and OPEN statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the PREPARE statement and dynamic SQL in Chapter 5.

PUT

Use the PUT statement to store a row in an insert buffer for later insertion into the database.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	It identifies a cursor	A DECLARE statement must have previously created the cursor.	Identifier, p. 1-723
cursor variable	Host variable that identifies a cursor	Host variable must be a character data type. A DECLARE statement must have previously created the cursor.	Variable name must conform to language-specific rules for variable names.

(1 of 2)

Element	Purpose	Restrictions	Syntax
descriptor	Quoted string that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-757
descriptor variable	Host variable name that identifies the system-descriptor area	System-descriptor area must already be allocated.	Quoted String, p. 1-757
indicator variable	Host variable that receives a return code if null data is placed in the corresponding data variable	Variable cannot be a DATETIME or INTERVAL data type.	Variable name must conform to language-specific rules for variable names.
sqlda pointer	It points to an **sqlda** structure that defines the type and memory location of values that correspond to the question-mark (?) placeholder in a prepared statement	You cannot begin an **sqlda** pointer with a dollar sign ($) or a colon (:).	Prepared statement information stored in *sqlda* structure by a DESCRIBE statement, p. 1-255.
variable name	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Variable must be a character data type.	Variable name must conform to language-specific rules for variable names.

(2 of 2)

Usage

Each PUT statement stores a row in an insert buffer that was created when *cursor name* was opened. If the buffer has no room for the new row when the statement executes, the buffered rows are written to the database in a block and the buffer is emptied. As a result, some PUT statement executions cause rows to be written to the database, and some do not.

You can use the FLUSH statement to write buffered rows to the database without adding a new row. The CLOSE statement writes any remaining rows before it closes an insert cursor.

If the current database uses explicit transactions, you must execute a PUT statement within a transaction.

The following example uses a PUT statement in INFORMIX-ESQL/C:

```
EXEC SQL prepare ins_mcode from
    'insert into manufact values(?,?)';
EXEC SQL declare mcode cursor for ins_mcode;
EXEC SQL open mcode;
EXEC SQL put mcode from :the_code, :the_name;
```

X/O

PUT is not an X/Open SQL statement. Therefore, you get a warning message if you compile a PUT statement in X/Open mode in an SQL API product. For details on compiling in X/Open mode, see your SQL API product manual.♦

Supplying Inserted Values

The values that reside in the inserted row can come from one of the following sources:

- Constant values that are written into the INSERT statement
- Program variables that are named in the INSERT statement
- Program variables that are named in the FROM clause of the PUT statement
- Values that are prepared in memory addressed by an **sqlda** structure or a system-descriptor area and then named in the USING clause of the PUT statement

Using Constant Values in INSERT

The VALUES clause of the INSERT statement lists the values of the inserted columns. One or more of these values might be constants (that is, numbers or character strings).

When *all* the inserted values are constants, the PUT statement has a special effect. Instead of creating a row and putting it in the buffer, the PUT statement merely increments a counter. When you use a FLUSH or CLOSE statement to empty the buffer, one row and a repetition count are sent to the database server, which inserts that number of rows.

In the following INFORMIX-ESQL / C example, 99 empty customer records are inserted into the **customer** table. Because all values are constants, no disk output occurs until the cursor closes. (The constant zero for **customer_num** causes generation of a SERIAL value.)

```
int count;
EXEC SQL declare fill_c cursor for
    insert into customer(customer_num) values(0);
EXEC SQL open fill_c;
for (count = 1; count <= 99; ++count)
    EXEC SQL put fill_c;
EXEC SQL close fill_c;
```

Naming Program Variables in INSERT

When the INSERT statement is written as part of the cursor declaration (in the DECLARE statement), you can name program variables in the VALUES clause. When each PUT statement is executed, the contents of the program variables at that time are used to populate the row that is inserted into the buffer.

If you are creating an insert cursor (using DECLARE with INSERT), you must use only program variables in the VALUES clause. Variable names are not recognized in the context of a prepared statement; you associate a prepared statement with a cursor through its statement identifier.

The following INFORMIX-ESQL / C example illustrates the use of an insert cursor. The code includes the following statements:

- The DECLARE statement associates a cursor called **ins_curs** with an INSERT statement that inserts data into the **customer** table. The VALUES clause names a data structure that is called **cust_rec**; the ESQL / C preprocessor converts **cust_rec** to a list of values, one for each component of the structure.
- The OPEN statement creates a buffer.
- A function that is not defined in the example obtains customer information from an interactive user and leaves it in **cust_rec**.
- The PUT statement composes a row from the current contents of the **cust_rec** structure and sends it to the row buffer.

- The CLOSE statement inserts into the **customer** table any rows that remain in the row buffer and closes the insert cursor.

```
int keep_going = 1;
EXEC SQL BEGIN DECLARE SECTION
    struct cust_row { /* fields of a row of customer table */ } cust_rec;
EXEC SQL END DECLARE SECTION

EXEC SQL declare ins_curs cursor for
        insert into customer values (:cust_row);
EXEC SQL open ins_curs;
for (; (sqlca.sqlcode == 0) && (keep_going) ;)
    {
    keep_going = get_user_input(cust_rec); /* ask user for new customer */
    if (keep_going )                       /* user did supply customer info */
        {
        cust_rec.customer_num = 0;         /* request new serial value */
        EXEC SQL put ins_curs;
        }
    if (sqlca.sqlcode == 0)                /* no error from PUT */
        keep_going = (prompt_for_y_or_n("another new customer") =='Y')
    }
EXEC SQL close ins_curs;
```

Naming Program Variables in PUT

When the INSERT statement is prepared (see the PREPARE statement on page 1-402), you cannot use program variables in its VALUES clause. However, you can represent values using a question-mark (?) placeholder. List the names of program variables in the FROM clause of the PUT statement to supply the missing values. The following INFORMIX-ESQL / C example lists host variables in a PUT statement:

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
    char ins_comp[80];
    char u_company[20];
EXEC SQL END DECLARE SECTION;

main()
{
    EXEC SQL connect to 'stores7';
    EXEC SQL prepare ins_comp from
        'insert into customer (customer_num, company) values (0, ?)';
    EXEC SQL declare ins_curs cursor for ins_comp;
    EXEC SQL open ins_curs;

    while (1)
        {
        printf("\nEnter a customer: ");
        gets(u_company);
        EXEC SQL put ins_curs from :u_company;
        printf("Enter another customer (y/n) ? ");
```

```
        if (answer = getch() != 'y')
            break;
        }
    EXEC SQL close ins_curs;
    EXEC SQL disconnect all;
}
```

Using a System-Descriptor Area

You can create a system-descriptor area that describes the data type and memory location of one or more values. You can then specify that system-descriptor area in the USING SQL DESCRIPTOR clause of the PUT statement.

For details on using descriptors, see your SQL API manual. The following INFORMIX-ESQL / C and INFORMIX-ESQL / COBOL examples show how to associate values from a system-descriptor area:

INFORMIX-ESQL/C

```
EXEC SQL put selcurs using sql descriptor 'desc1';
```

INFORMIX-ESQL/COBOL

```
EXEC SQL PUT SEL_CURS USING SQL DESCRIPTOR 'DESC1' END-EXEC.
```

Using an sqlda Structure

E/C

You can create an **sqlda** structure that describes the data type and memory location of one or more values. Then you can specify the **sqlda** structure in the USING DESCRIPTOR clause of the PUT statement. Each time the PUT statement executes, the values that the **sqlda** structure describes are used to replace question-mark (?) placeholders in the INSERT statement. This process is similar to using a FROM clause with a list of variables, except that your program has full control over the memory location of the data values.

The following example shows the usage of the PUT ... USING DESCRIPTOR clause. For details on the **sqlda** structure, see the *INFORMIX-ESQL/C Programmer's Manual*.

```
EXEC SQL put selcurs using descriptor pointer2;
```

♦

Writing Buffered Rows

When the OPEN statement opens an insert cursor, an insert buffer is created. The PUT statement puts a row into this insert buffer. The block of buffered rows is inserted into the database table as a block only when necessary; this process is called *flushing the buffer*. The buffer is flushed after any of the following events:

- The buffer is too full to hold the new row at the start of a PUT statement.
- A FLUSH statement executes.
- A CLOSE statement closes the cursor.
- An OPEN statement executes, naming the cursor.

  When the OPEN statement is applied to an open cursor, it closes the cursor before reopening it; this implied CLOSE statement flushes the buffer.
- A COMMIT WORK statement executes.
- The buffer contains blob data (flushed after a single PUT statement).

If the program terminates without closing an insert cursor, the buffer remains unflushed. Rows that were inserted into the buffer since the last flush are lost. Do not rely on the end of the program to close the cursor and flush the buffer.

Error Checking

The **sqlca** contains information on the success of each PUT statement as well as information that lets you count the rows that were inserted. The result of each PUT statement is contained in the fields of the **sqlca**, as the following table shows.

ESQL/C	ESQL/COBOL
sqlca.sqlcode, SQLCODE	SQLCODE OF SQLCA
sqlca.sqlerrd[2]	SQLERRD[3] OF SQLCA

Data buffering with an insert cursor means that errors are not discovered until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is flushed. When an error is discovered, rows in the buffer that are located after the error are *not* inserted; they are lost from memory.

The SQLCODE field is set to 0 if no error occurs; otherwise, it is set to an error code. The third element of the **sqlerrd** array is set to the number of rows that are successfully inserted into the database:

- If a row is put into the insert buffer, and buffered rows are *not* written to the database, SQLCODE and **sqlerrd** are set to 0 (SQLCODE because no error occurred, and **sqlerrd** because no rows were inserted).
- If a block of buffered rows is written to the database during the execution of a PUT statement, SQLCODE is set to 0 and **sqlerrd** is set to the number of rows that was successfully inserted into the database.
- If an error occurs while the buffered rows are written to the database, SQLCODE indicates the error, and **sqlerrd** contains the number of successfully inserted rows. (The uninserted rows are discarded from the buffer.)

Tip: *When you encounter an SQLCODE error, a corresponding SQLSTATE error value might exist. Check the GET DIAGNOSTICS statement for information about how to get the SQLSTATE value and how to use the GET DIAGNOSTICS statement to interpret the SQLSTATE value.*

Counting Total and Pending Rows

To count the number of rows that were actually inserted in the database and the number not yet inserted, perform the following procedure:

- Prepare two integer variables (for example, **total** and **pending)**.
- When the cursor is opened, set both variables to 0.
- Each time a PUT statement executes, increment both **total** and **pending**.
- Whenever a PUT or FLUSH statement executes, or the cursor closes, subtract the third field of the **SQLERRD** array from **pending**.

At any time, (**total** - **pending)** represents the number of rows that were actually inserted. If all commands are successful, **pending** contains zero after the cursor is closed. If an error occurs during a PUT, FLUSH, or CLOSE statement, the value that remains in **pending** is the number of uninserted (discarded) rows.

References

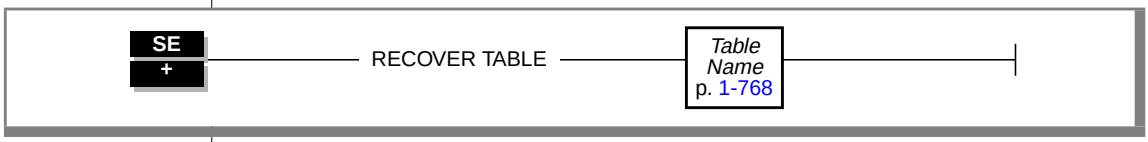
See the CLOSE, FLUSH, DECLARE, and OPEN statements, which are cursor-related, in this manual. Also see the ALLOCATE DESCRIPTOR statement.

In the *Informix Guide to SQL: Tutorial*, see the discussion of the PUT statement in Chapter 6.

RECOVER TABLE

Use the RECOVER TABLE statement with INFORMIX-SE to restore a database table in the event of failure.

Syntax



Usage

The RECOVER TABLE statement applies the table audit trail to a backup copy of the database. INFORMIX-SE uses audit trails to record operations on a per-table basis. You can issue a RECOVER TABLE statement if you own the table or have the DBA privilege for the database.

If a system failure occurs, use an operating-system utility to restore each table file for which you have an audit trail. Issue the RECOVER TABLE statement to update each newly restored table with the transactions that are recorded in the audit trail.

Backup/Restore Procedure

The recommended backup / restore procedure for making backup copies of a database that includes audit trails is described in the following list:

- Execute the DROP AUDIT statement for each table that has an audit trail. The DROP AUDIT statement ends system logging to the audit-trail files.
- Execute the CREATE AUDIT statement for each table, specifying the pathname of the new audit trail. For maximum protection, specify a location that is not on the same storage device as the database. You can also select a filename that reflects the table name and the sequence of the file in the audit trail (for example, **audit_cust_001** or **audit_cust_002)**. The CREATE AUDIT statement registers the new name and location of the audit-trail file in the **systables** system catalog table.
- Use an operating-system utility to back up the database files.

During execution, the RECOVER TABLE statement checks that the audit trail and table name have consistent record numbers for rows where changes occurred. In extremely rare instances, the RECOVER TABLE statement can find an inconsistency that a system crash caused. In this case only, the RECOVER TABLE statement stops, and you must restore the table manually.

The following list of actions and statements serves as a guide to recover the **customer** table. First, restore the **customer** table from your last archive copy. Second, run the following statements, which assume that your audit trail began immediately after you created the archive copy:

```
RECOVER TABLE customer
DROP AUDIT FOR customer
CREATE AUDIT FOR customer
```

Third, create a new backup of the recovered table.

The audit-trail file is not in human-readable form. Even so, the DBA can copy the file to a database (**.dat**) file and manipulate it. The modified file can be copied back to the audit-trail file, enabling customized restorations of particular tables. For example, you can modify the audit-trail file to exclude rows that a particular user entered or to undo specific transactions. For specific instructions on modifying audit-trail files, refer to the manual for your application-development tool.

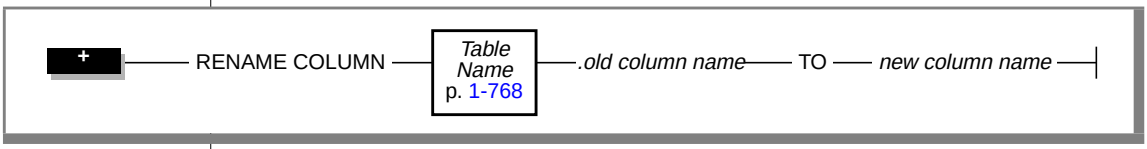
References

See the CREATE AUDIT and DROP AUDIT statements in this manual.

RENAME COLUMN

Use the RENAME COLUMN statement to change the name of a column.

Syntax



Element	Purpose	Restrictions	Syntax
new column name	The new name to be assigned to the column	The new name of the column must be unique within the table. If you rename a column that appears within a trigger definition, the new column name replaces the old column name in the trigger definition only if certain conditions are met. See "How Triggers Are Affected" on page 1-429 for more information on this restriction.	Identifier, p. 1-723
.old column name	The current name of the column you want to rename	The column must exist within the table. The column name must be preceded by a period. You can put a space between the table name and *.old column name*, or you can omit the space.	Identifier, p. 1-723

Usage

You can rename a column of a table if any of the following conditions are true:

- You own the table.
- You have the DBA privilege on the database.
- You have the Alter privilege on the table.

When you rename a column, choose a column name that is unique within the table.

SE

You cannot use a ROLLBACK WORK statement to undo a RENAME COLUMN statement that successfully executes. If you roll back a transaction that contains a RENAME COLUMN statement, the column retains its new name, and you do not receive an error message. ♦

How Views and Check Constraints Are Affected

If you rename a column that a view in the database references, the text of the view in the **sysviews** system catalog table is updated to reflect the new column name.

If you rename a column that a check constraint in the database references, the text of the check constraint in the **syschecks** system catalog table is updated to reflect the new column name.

How Triggers Are Affected

If you rename a column that appears within a trigger, it is replaced with the new name only in the following instances:

- When it appears as part of a correlation name inside the FOR EACH ROW action clause of a trigger
- When it appears as part of a correlation name in the INTO clause of an EXECUTE PROCEDURE statement
- When it appears as a triggering column in the UPDATE clause

When the trigger executes, if the database server encounters a column name that no longer exists in the table, it returns an error

Example of RENAME COLUMN

The following example assigns the new name of **c_num** to the **customer_num** column in the **customer** table:

```
RENAME COLUMN customer.customer_num TO c_num
```

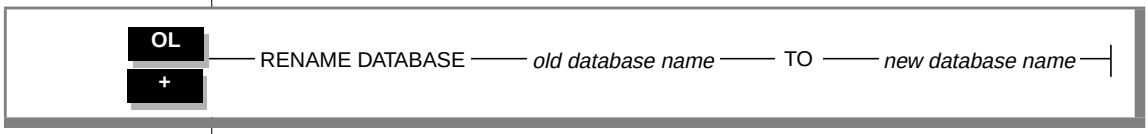
References

See the ALTER TABLE, CREATE TABLE, and RENAME TABLE statements in this manual.

RENAME DATABASE

Use the RENAME DATABASE statement to change the name of a database.

Syntax



Element	Purpose	Restrictions	Syntax
new database name	The new name that you want to assign to the database	Name must be unique. You cannot rename the current database. The database to be renamed must not be opened by any users when the RENAME DATABASE command is issued.	Database Name, p. 1-660
old database name	The name of the database that you want to rename	The database name must exist.	Database Name, p. 1-660

Usage

You can rename a database if *either* of the following statements is true:

- You created the database.
- You have the DBA privilege on the database.

You can only rename local databases. You can rename a local database from inside a stored procedure.

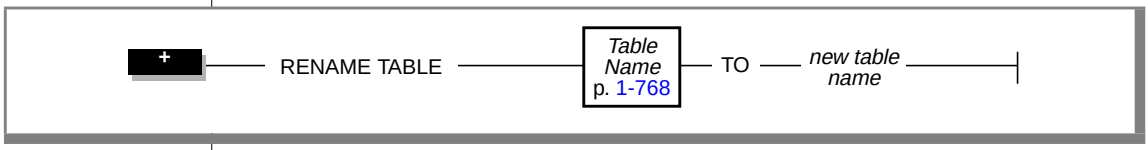
References

See the CREATE DATABASE statement in this manual.

RENAME TABLE

Use the RENAME TABLE statement to change the name of a table.

Syntax



Element	Purpose	Restrictions	Syntax
new table name	The new name that you want to assign to the table	You cannot use the *owner.* convention in the new name of the table.	Identifier, p. 1-723

Usage

You can rename a table if any of the following statements are true:

- You own the table.
- You have the DBA privilege on the database.
- You have the Alter privilege on the table.

You cannot change the table owner by renaming the table. You can use the *owner.* convention in the old name of the table, but an error occurs during compilation if you try to use the *owner.* convention in the new name of the table.

ANSI

In an ANSI-compliant database, you must use the *owner.* convention in the old name of the table if you are referring to a table that you do not own. ♦

You cannot use the RENAME TABLE statement to move a table from the current database to another database or to move a table from another database to the current database. The table that you want to rename must reside in the current database. The renamed table that results from the statement remains in the current database.

SE

You cannot use a ROLLBACK WORK statement to undo a RENAME TABLE statement that successfully executes. If you roll back a transaction that contains a RENAME TABLE statement, the table retains its new name, and you do not receive an error message. ♦

Renaming Tables That Views Reference

If a view references the table that was renamed, and the view resides in the same database as the table, the database server updates the text of the view in the **sysviews** system catalog table to reflect the new table name. See the *Informix Guide to SQL: Reference* for further information on the **sysviews** system catalog table.

Renaming Tables That Have Triggers

If you rename a table that has a trigger, it produces the following results:

- The database server replaces the name of the table in the trigger definition.
- The table name is *not* replaced where it appears inside any triggered actions.
- The database server returns an error if the new table name is the same as a correlation name in the REFERENCING clause of the trigger definition.

When the trigger executes, the database server returns an error if it encounters a table name for which no table exists.

Example of Renaming a Table

The following example reorganizes the **items** table. The intent is to move the **quantity** column from the fifth position to the third. The example illustrates the following steps:

1. Create a new table, **new_table**, that contains the column **quantity** in the third position.
2. Fill the table with data from the current **items** table.
3. Drop the old **items** table.
4. Rename **new_table** with the name **items**.

The following example uses the RENAME TABLE statement as the last step:

```
CREATE TABLE new_table
    (
    item_num    SMALLINT,
    order_num   INTEGER,
    quantity    SMALLINT,
    stock_num   SMALLINT,
    manu_code   CHAR(3),
    total_price MONEY(8)
    )
INSERT INTO new_table
    SELECT item_num, order_num, quantity, stock_num,
           manu_code, total_price
    FROM items
DROP TABLE items
RENAME TABLE new_table TO items
```

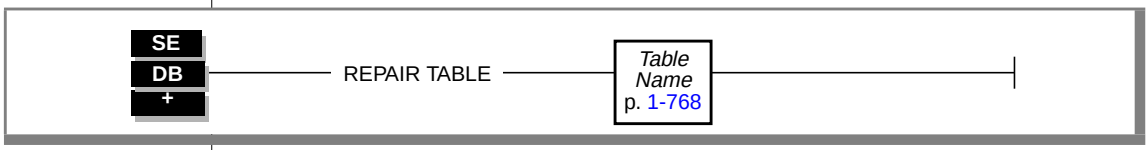
References

See the ALTER TABLE, CREATE TABLE, DROP TABLE, and RENAME COLUMN statements in this manual.

REPAIR TABLE

Use the REPAIR TABLE statement to remove and rebuild table indexes or data that might have been damaged or corrupted because of a power failure, computer crash, or other unexpected program stoppage. Only damaged tables are rebuilt. To determine whether you need to use the REPAIR TABLE statement, you can first issue the CHECK TABLE statement.

Syntax



Usage

Specify the name of the table for which you want to restore the integrity of the index files, as the following example shows:

```
REPAIR TABLE cust_calls
```

You must specify a table that is in a database in the current directory. If you specify a simple name for a database in the DATABASE command, but the database is not located in the current directory, REPAIR TABLE does not search the **DBPATH** environment variable to find the directory for the database; the REPAIR TABLE statement will fail. Similarly, if you specify an explicit pathname for a database in the DATABASE command, but the database is not located in the current directory, REPAIR TABLE does not search for the database in the specified directory; the REPAIR TABLE statement will fail.

You cannot use the REPAIR TABLE statement on a table unless you own it or have the DBA privilege on the database. You cannot use the REPAIR TABLE statement on the system catalog table **systables** unless you have the DBA privilege on the database.

The REPAIR TABLE statement calls the **secheck** utility.

References

See the CHECK TABLE statement in this manual.

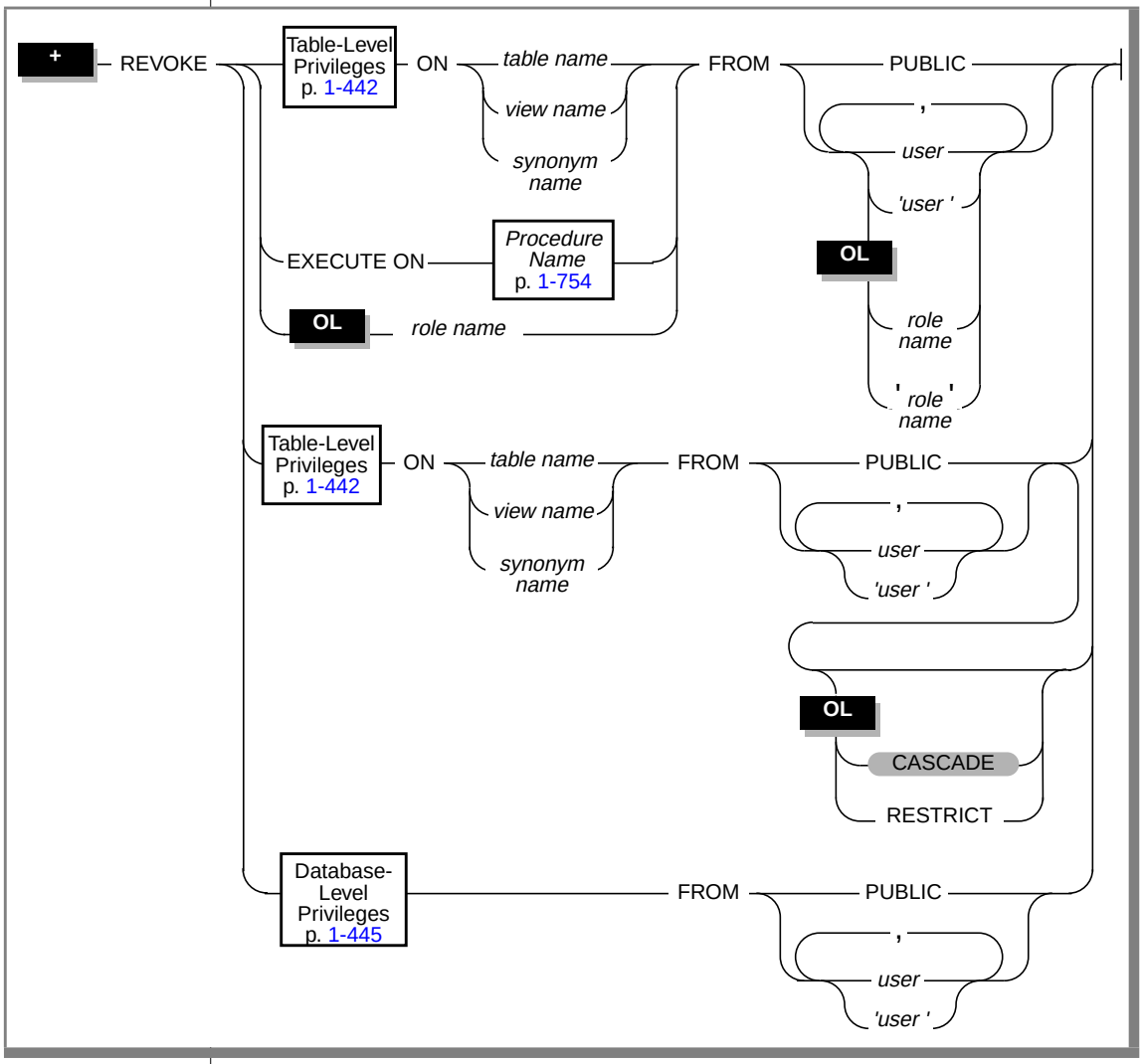
See the *INFORMIX-SE Administrator's Guide* for a full description of **secheck.**

REVOKE

You can use the REVOKE statement for the following purposes:

- You can revoke privileges on a table or view from a user or a role.
- You can revoke the privilege to execute a procedure from a user or a role.
- You can revoke privileges on a database from a user.
- You can revoke a role from a user or from another role.

Syntax


+
REVOKE
Table-Level Privileges p. 1-442
ON
table name
view name
synonym name
FROM
PUBLIC
,
user
'user '
EXECUTE ON
Procedure Name p. 1-754
OL
role name
OL
role name
' role name '
Table-Level Privileges p. 1-442
ON
table name
view name
synonym name
FROM
PUBLIC
,
user
'user '
OL
CASCADE
RESTRICT
Database-Level Privileges p. 1-445
FROM
PUBLIC
,
user
'user '

Element	Purpose	Restriction	Syntax
role name	Names the role from which a privilege or another role is to be revoked or names the role to be revoked from a user or another role.	The role must have been created with the CREATE ROLE statement and granted with the GRANT statement	Identifier, p. 1-723
synonym name	The synonym name for which a privilege is revoked	The name must be an existing synonym name.	Synonym Name, p. 1-766
table name	The table name for which a privilege is revoked	The name must be an existing table name.	Table Name, p. 1-768
user	Names the user from whom a privilege or role is revoked	The user must be a valid user.	Identifier, p. 1-723
view name	The view name for which a privilege is revoked	The name must be an existing view name.	View Name, p. 1-772

Usage

You can use the REVOKE statement with the GRANT statement to control finely the ability of users to modify the database as well as to access and modify data in the tables.

If you use the PUBLIC keyword after the FROM keyword, the REVOKE statement revokes privileges from all users.

You can revoke all or some of the privileges that you granted to other users. No one can revoke privileges that another user grants.

If you revoke the EXECUTE privilege on a stored procedure from a user, that user can no longer run that procedure using either the EXECUTE PROCEDURE or CALL statements.

If you use quotes, *user* appears exactly as typed.

ANSI

In an ANSI-compliant database, if you do not use quotes around *user*, the name of the user is stored in uppercase letters. ♦

Users cannot revoke privileges from themselves.

Using the REVOKE Statement with Roles

You can use the REVOKE statement to remove privileges from a role and remove a role from a user or another role. Once a role is revoked from a user, the user cannot enable that role. You can revoke all or some of the roles granted to a user or role.

If a role is revoked from a user, the privileges associated with the role cannot be acquired by the user with the SET ROLE statement. However, this does not affect the currently acquired privileges.

You can use the REVOKE statement to revoke table-level privileges from a role; however, you cannot use the RESTRICT or CASCADE clauses when you do so.

Only the DBA or a user granted a role with the WITH GRANT OPTION can revoke privileges for a role.

If you revoke the Execute privilege on a stored procedure from a role, that role can no longer run that procedure.

Users cannot revoke roles from themselves. When you revoke a role, you cannot revoke the WITH GRANT OPTION separately. If the role was granted with the WITH GRANT OPTION, both the role and grant option are revoked.

The following example revokes the **engineer** role from the user **maryf**:

```
REVOKE engineer FROM maryf
```

Revoking Privileges Granted from WITH GRANT OPTION

If you revoke from *user* the privileges that you granted using the WITH GRANT OPTION keywords, you sever the chain of privileges granted by that *user*. In this case, when you revoke privileges from *user*, you automatically revoke the privileges of all users who received privileges from *user* or from the chain that *user* created. You can also specify this default condition with the CASCADE keyword.

Controlling the Scope of a REVOKE with the RESTRICT Option

Use the RESTRICT keyword to control the success or failure of the REVOKE command based on the existence of dependencies on the objects that are being revoked. The following list shows the dependencies that cause the REVOKE statement to fail when you use the RESTRICT keyword:

- The user from whom the privilege is to be revoked has granted this privilege to another user or users.
- A view depends on a Select privilege that is being revoked.
- A foreign-key constraint depends on a References privilege that is being revoked.

Failure of the REVOKE When the Revokee Has Granted a Privilege

A REVOKE statement with the RESTRICT keyword fails if the user from whom a privilege is being revoked has granted the same privilege to another user or users. However, the same REVOKE statement does not fail if the revokee has the right to grant the privilege to other users but has not actually granted the privilege to any other user. We can illustrate these points by means of examples.

Assume that the user **clara** has granted the Select privilege on the **customer** table to the user **ted**, and she has also granted user **ted** the right to grant the Select privilege to other users. User **ted** has used this authority to grant the Select privilege on the **customer** table to the user named **tania**. Now user **clara** attempts to revoke the Select privilege from user **ted** with the following REVOKE statement:

```
REVOKE SELECT ON customer FROM ted RESTRICT
```

This statement fails because user **ted** has granted the Select privilege to user **tania**.

What if the revokee has the right to grant the privilege to other users but has not actually granted this privilege to any other user? For example, assume that the user **clara** has granted the Select privilege on the **customer** table to the user **roger**, and she has also granted user **roger** the right to grant the Select privilege to other users. However, user **roger** has not used this authority to granted the Select privilege to any other user. Now user **clara** attempts to revoke the Select privilege from user **roger** with the following REVOKE statement:

```
REVOKE SELECT ON customer FROM roger RESTRICT
```

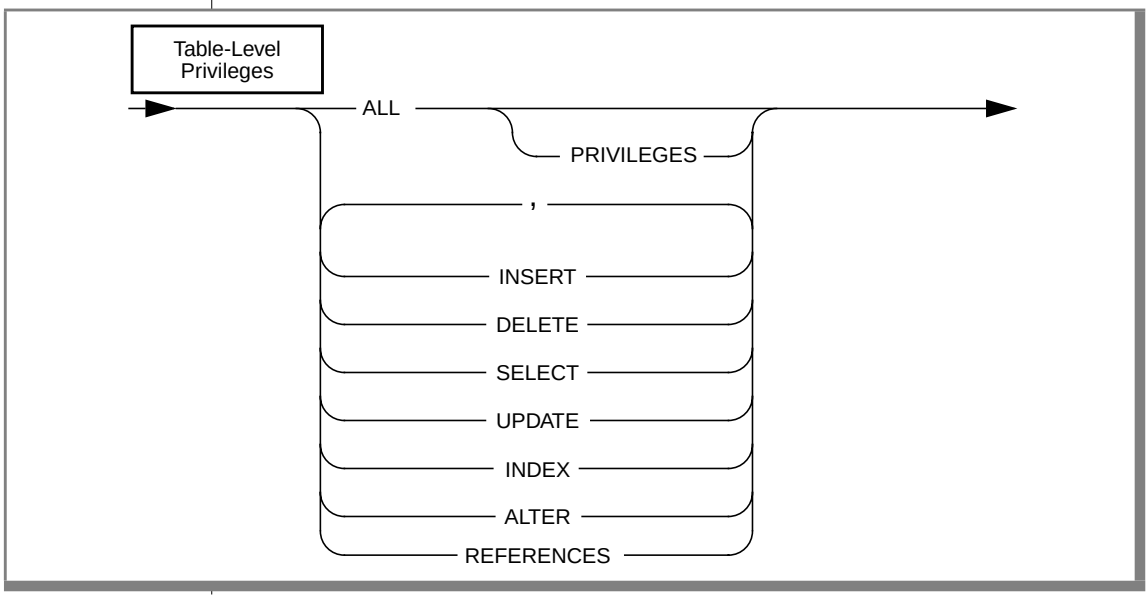
This statement succeeds because user **roger** has not granted the Select privilege to any other user.

REVOKE and ROLLBACK WORK

SE

You cannot use a ROLLBACK WORK statement to undo a REVOKE statement that successfully executes. If you roll back a transaction that contains a REVOKE statement, the privilege is not granted again to the user, and you do not receive an error message. ♦

Table-Level Privileges



To revoke a table-level privilege from a user, you must revoke all occurrences of the privilege. For example, if two users grant the same privilege to a user, both of them must revoke the privilege. If one grantor revokes the privilege, the user retains the privilege received from the other grantor. (The database server keeps a record of each table-level grant in the **syscolauth** and **systabauth** system catalog tables.)

If a table owner grants a privilege to PUBLIC, the owner cannot revoke the same privilege from any particular user. For example, if the table owner grants the Select privilege to PUBLIC and then attempts to revoke the Select privilege from **mary**, the REVOKE statement generates an error. The Select privilege was granted to PUBLIC, not to **mary**, and therefore the privilege cannot be revoked from **mary**. (ISAM error number 111, `No record found`, refers to the lack of a record in either the **syscolauth** or **systabauth** system catalog table, which would represent the grant that the table owner now wants to revoke.)

You can revoke table-level privileges individually or in combination. List the keywords that correspond to the privileges that you are revoking from *user*. The keywords are described in the following list. Unlike the GRANT statement, the REVOKE statement does not allow you to qualify the Select, Update, or References privilege with a column name. Thus you cannot revoke access on specific columns.

Privilege	Functions
INSERT	Provides the ability to insert rows
DELETE	Provides the ability to delete rows
SELECT	Provides the ability to display data obtained from a SELECT statement
UPDATE	Provides the ability to change column values
INDEX	Provides the ability to create permanent indexes. You must have the Resource privilege to take advantage of the Index privilege. (Any user with the Connect privilege can create indexes on temporary tables.)

(1 of 2)

Privilege	Functions
ALTER	Provides the ability to add or delete columns, modify column data types, and add or delete constraints. This privilege also provides the ability to set the object mode of unique indexes and constraints to the enabled, disabled, or filtering mode. In addition, this privilege provides the ability to set the object mode of non-unique indexes and triggers to the enabled or disabled modes.
REFERENCES	Provides the ability to reference columns in referential constraints. You must have the Resource privilege to take advantage of the References privilege. (However, you can add a referential constraint during an ALTER TABLE statement. This method does not require that you have the Resource privilege on the database.) Revoke the References privilege to disallow cascading deletes.
ALL	Provides all the preceding privileges. The PRIVILEGES keyword is optional.

(2 of 2)

Behavior of the ALL Keyword

The ALL keyword revokes all table-level privileges. If any or all of the table-level privileges do not exist for the revokee, the REVOKE statement with the ALL keyword executes successfully but returns the following SQLSTATE code:

```
01006 - Privilege not revoked
```

For example, assume that the user **hal** has the Select and Insert privileges on the **customer** table. User **jocelyn** wants to revoke all seven table-level privileges from user **hal**. So user **jocelyn** issues the following REVOKE statement:

```
REVOKE ALL ON customer FROM hal
```

This statement executes successfully but returns SQLSTATE code 01006.The SQLSTATE warning is returned with a successful statement as follows:

- The statement succeeds in revoking the Select and Insert privileges from user **hal** because user **hal** had those privileges.
- SQLSTATE code 01006 is returned because the other five privileges implied by the ALL keyword (the Delete, Update, References, Index,

Only a user with the DBA privilege can grant or revoke database-level privileges.

Three levels of database privileges control access. These privilege levels are, from lowest to highest, Connect, Resource, and DBA. To revoke a database privilege, specify one of the keywords CONNECT, RESOURCE, or DBA in the REVOKE statement.

Because of the hierarchical organization of the privileges (as outlined in the privilege definitions that are described later in this section), if you revoke either the Resource or the Connect privilege from a user with the DBA privilege, the statement has no effect. If you revoke the DBA privilege from a user who has the DBA privilege, the user retains the Connect privilege on the database. To deny database access to a user with the DBA or Resource privilege, you must first revoke the DBA or the Resource privilege and then revoke the Connect privilege in a separate REVOKE statement.

Similarly, if you revoke the Connect privilege from a user with the Resource privilege, the statement has no effect. If you revoke the Resource privilege from a user, the user retains the Connect privilege on the database.

The database privileges are associated with the following keywords.

Privilege	Functions
CONNECT	Lets you query and modify data. You can modify the database schema if you won the object that you want to modify. Any user with the Connect privilege can perform the following functions: ■ Connect to the database with the CONNECT statement or another connection statement ■ Execute SELECT, INSERT, UPDATE, and DELETE statements, provided that the user has the necessary table-level privileges ■ Create views, provided that the user has the Select privilege on the underlying tables ■ Create synonyms ■ Create temporary tables, and create indexes on the temporary tables ■ Alter or drop a table or an index, provided that the user owns the table or index (or has the Alter, Index, or References privilege on the table) ■ Grant privileges on a table, provided that the user owns the table (or has been given privileges on the table with the WITH GRANT OPTION keyword)
RESOURCE	Lets you extend the structure of the database. In addition to the capabilities of the Connect privilege, the holder of the Resource privilege can perform the following functions: ■ Create new tables ■ Create new indexes ■ Create new procedures

(1 of 2)

Privilege	Functions
DBA	Lets the holder of DBA privilege perform the following functions in addition to the capabilities of the Resource privilege: ■ Grant any database-level privilege, including the DBA privilege, to another user ■ Grant any table-level privilege to another user ■ Grant any table-level privilege to a role ■ Grant a role to a user or to another role ■ Execute the SET SESSION AUTHORIZATION statement ■ Use the NEXT SIZE keyword to alter extent sizes in the system catalog tables ■ Drop any object, regardless of who owns it ■ Create tables, views, and indexes as well as specify another user as owner of the objects ■ Execute the DROP DATABASE statement ■ Execute the START DATABASE and ROLLFORWARD DATABASE statements♦ ■ Insert, delete, or update rows of any system catalog table except **systables**

(2 of 2)

SE

Warning: *Although the user* ***informix*** *and DBAs can modify most system catalog tables (only the user* ***informix*** *can modify* ***systables****), Informix strongly recommends that you do not update, delete, or insert any rows in these tables. Modifying the system catalog tables can destroy the integrity of the database. Informix does support use of the* ALTER TABLE *statement to modify the size of the next extent of system catalog tables.*

References

See the GRANT, GRANT FRAGMENT, and REVOKE FRAGMENT statements in this manual.

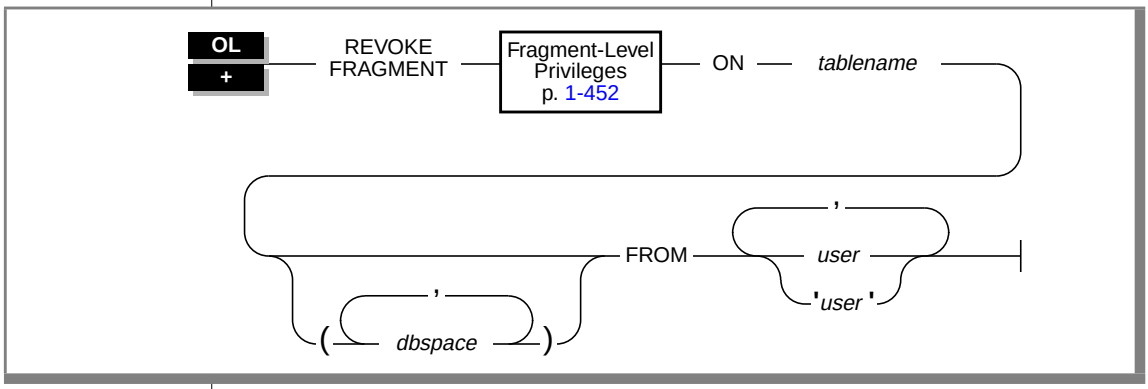
For information about roles, see the CREATE ROLE, DROP ROLE, and SET ROLE statements in this manual.

See the discussion of privileges and security in the *Informix Guide to SQL: Tutorial*.

REVOKE FRAGMENT

The REVOKE FRAGMENT statement enables you to revoke privileges that have been granted on individual fragments of a fragmented table. You can use this statement to revoke the Insert, Update, and Delete fragment-level privileges from users.

Syntax



Element	Purpose	Restrictions	Syntax
dbspace	The name of the dbspace where the fragment is stored. Use this parameter to specify the fragment or fragments on which privileges are to be revoked. If you do not specify a fragment, the REVOKE statement applies to all fragments in the specified table that have the specified privileges.	The specified dbspace or dbspaces must exist.	Identifier, p. 1-723
tablename	The name of the table that contains the fragment or fragments on which privileges are to be revoked. There is no default value.	The specified table must exist and must be fragmented by expression.	Table Name, p. 1-768
user	The name of the user or users from whom the specified privileges are to be revoked. There is no default value.	The user must be a valid user.	Identifier, p. 1-723

Usage

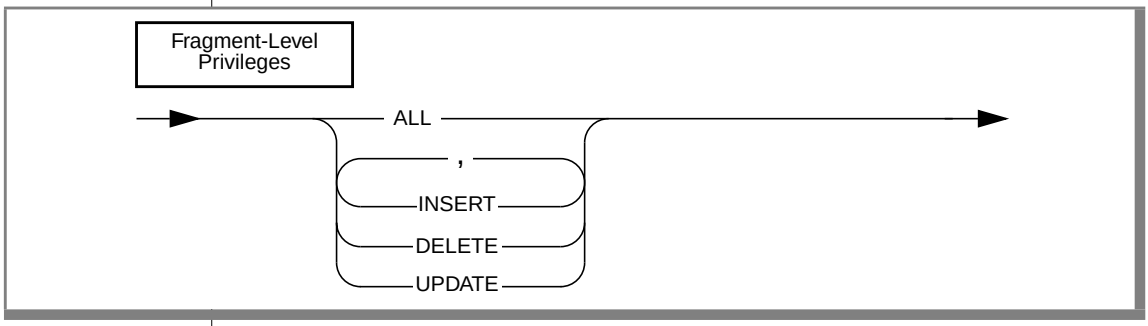
Use the REVOKE FRAGMENT statement to revoke the Insert, Update, or Delete privilege on one or more fragments of a fragmented table from one or more users.

The REVOKE FRAGMENT statement is only valid for tables that are fragmented according to an expression-based distribution scheme. See the ALTER FRAGMENT statement on page 1-22 for an explanation of expression-based distribution schemes.

You can specify one fragment or a list of fragments in the REVOKE FRAGMENT statement. To specify a fragment, name the dbspace in which the fragment resides.

You do not have to specify a particular fragment or a list of fragments in the REVOKE FRAGMENT statement. If you do not specify any fragments in the statement, the specified users lose the specified privileges on all fragments for which the users currently have those privileges.

Fragment-Level Privileges



You can revoke fragment-level privileges individually or in combination. List the keywords that correspond to the privileges that you are revoking from *user*. The keywords are described in the following list.

Privilege	Functions
ALL	Revokes all privileges currently granted on a table fragment
INSERT	Revokes Insert privilege on a table fragment. This privilege gives the user the ability to insert rows in the fragment.
DELETE	Revokes Delete privilege on a table fragment. This privilege gives the user the ability to delete rows in the fragment.
UPDATE	Revokes Update privilege on a table fragment. This privilege gives the user the ability to update rows in the fragment and to name any column of the table in an UPDATE statement.

If you specify the ALL keyword in a REVOKE FRAGMENT statement, the specified users lose all fragment-level privileges that they currently have on the specified fragments.

For example, assume that a user currently has the Update privilege on one fragment of a table. If you use the ALL keyword to revoke all current privileges on this fragment from this user, the user loses the Update privilege that he or she had on this fragment.

Examples of the REVOKE FRAGMENT Statement

The examples that follow are based on the **customer** table. All the examples assume that the **customer** table is fragmented by expression into three fragments that reside in the dbspaces that are named **dbsp1**, **dbsp2**, and **dbsp3**.

Revoking One Privilege

The following statement revokes the Update privilege on the fragment of the **customer** table in **dbsp1** from the user **ed**:

```
REVOKE FRAGMENT UPDATE ON customer (dbsp1) FROM ed
```

Revoking More Than One Privilege

The following statement revokes the Update and Insert privileges on the fragment of the **customer** table in **dbsp1** from the user **susan**:

```
REVOKE FRAGMENT UPDATE, INSERT ON customer (dbsp1) FROM susan
```

Revoking All Privileges

The following statement revokes all privileges currently granted to the user **harry** on the fragment of the **customer** table in **dbsp1**.:

```
REVOKE FRAGMENT ALL ON customer (dbsp1) FROM harry
```

Revoking Privileges on More Than One Fragment

The following statement revokes all privileges currently granted to the user **millie** on the fragments of the **customer** table in **dbsp1** and **dbsp2**:

```
REVOKE FRAGMENT ALL ON customer (dbsp1, dbsp2) FROM millie
```

Revoking Privileges from More Than One User

The following statement revokes all privileges currently granted to the users **jerome** and **hilda** on the fragment of the **customer** table in **dbsp3**:

```
REVOKE FRAGMENT ALL ON customer (dbsp3) FROM jerome, hilda
```

Revoking Privileges Without Specifying Fragments

The following statement revokes all current privileges from the user **mel** on all fragments for which this user currently has privileges:

```
REVOKE FRAGMENT ALL ON customer FROM mel
```

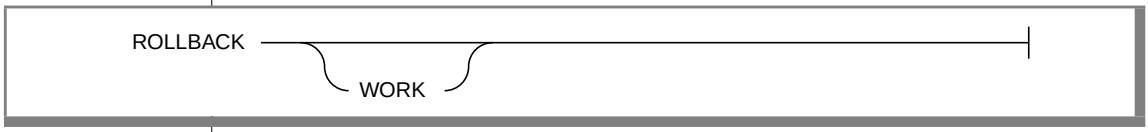
References

See the REVOKE and GRANT FRAGMENT statements in this manual.

ROLLBACK WORK

Use the ROLLBACK WORK statement to cancel a transaction and undo any changes that occurred since the beginning of the transaction.

Syntax



Usage

The ROLLBACK WORK statement is valid only in databases with transactions.

In a database that is not ANSI-compliant, start a transaction with a BEGIN WORK statement. You can end a transaction with a COMMIT WORK statement or cancel the transaction with a ROLLBACK WORK statement. The ROLLBACK WORK statement restores the database to the state that existed before the transaction began. Use the ROLLBACK WORK statement only at the end of a multistatement operation.

The ROLLBACK WORK statement releases all row and table locks that the cancelled transaction holds. If you issue a ROLLBACK WORK statement when no transaction is pending, an error occurs.

ANSI

In an ANSI-compliant database, transactions are implicit. Transactions start after each COMMIT WORK or ROLLBACK WORK statement. If you issue a ROLLBACK WORK statement when no transaction is pending, the statement is accepted but has no effect. ♦

SE

A ROLLBACK WORK statement undoes all database changes except those that result from GRANT or REVOKE statements or from data definition statements. Data definition statements are treated as single transactions. If they execute successfully, they are committed automatically and cannot be rolled back by the ROLLBACK WORK statement. Data definition statements include statements that modify the number, names, or indexes of tables and statements that modify the number, names, or data types of columns. For a list of data definition statements, see “Data Definition Statements” on page 1-12.

If a transaction rolls back, the actions that are taken to undo the transaction are also logged to table audit trails, if any exist. ♦

ESQL

The ROLLBACK WORK statement closes all open cursors except those that are declared with hold, which remain open despite transaction activity.

If you use the ROLLBACK WORK statement within a routine that a WHENEVER statement calls, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. This prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning. ♦

References

See the BEGIN WORK and COMMIT WORK statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of ROLLBACK WORK in Chapter 5.

ROLLFORWARD DATABASE

Use the ROLLFORWARD DATABASE statement with the INFORMIX-SE database server to apply the transaction log file to a restored database.

Syntax

Usage

To restore a database, you need both the archive copy of the database and the transaction log that began immediately after the archive copy was made.

To execute the ROLLFORWARD DATABASE statement, you need the DBA privilege. Always precede a ROLLFORWARD DATABASE statement with a CLOSE DATABASE statement. The ROLLFORWARD DATABASE statement fails if a database is open.

The ROLLFORWARD DATABASE statement sets an exclusive lock on the database to prevent access by other processes. If another process is using the database (even if the database is only being read), the ROLLFORWARD DATABASE statement fails.

The database remains locked after the ROLLFORWARD DATABASE statement executes. This allows you to check for errors before you give access to other users. When you are satisfied that the database is ready for use, issue the CLOSE DATABASE statement to release the exclusive lock. You can open the database with the DATABASE statement.

You must be working on a database server to issue a ROLLFORWARD DATABASE statement. You cannot execute the statement from a client computer.

References

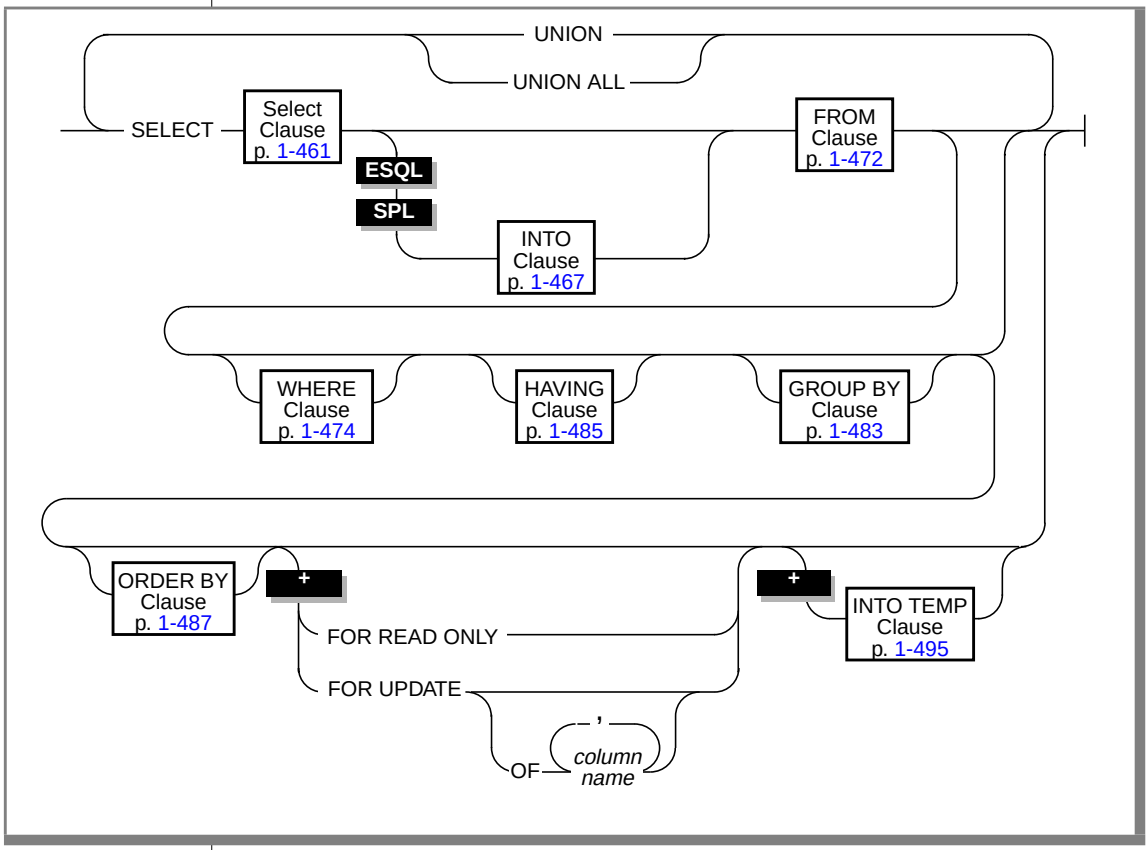
See the BEGIN WORK, COMMIT WORK, CLOSE DATABASE, DATABASE, and ROLLBACK WORK statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of backups and logs in Chapter 4.

SELECT

Use the SELECT statement to query a database.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column that can be updated after a fetch	The specified column must be in the table, but it does not have to be in the select list of the SELECT clause.	Identifier, p. 1-723

Usage

You can query the tables in the current database, a database that is not current, or a database that is on a different database server from your current database.

SE

You can query only the current database. ♦

The SELECT statement comprises many basic clauses. Each clause is described in the following list.

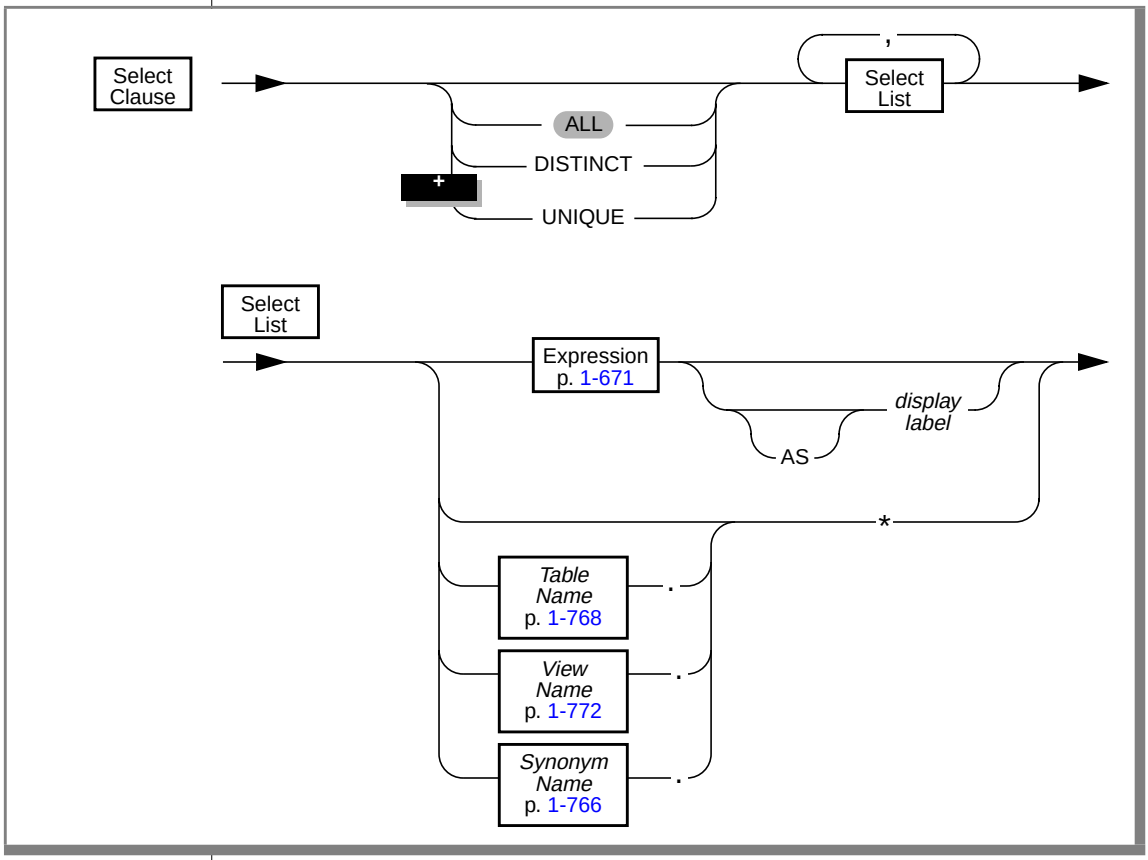
Clause	Purpose
SELECT	Names a list of items to be read from the database
INTO	Specifies the program variables or host variables that receive the selected data ♦
FROM	Names the tables that contain the selected columns
WHERE	Sets conditions on the selected rows
GROUP BY	Combines groups of rows into summary results
HAVING	Sets conditions on the summary results
ORDER BY	Orders the selected rows
INTO TEMP	Creates a temporary table in the current database and puts the results of the query into the table
FOR UPDATE	Specifies that the values returned by the SELECT statement can be updated after a fetch
FOR READ ONLY	Specifies that the values returned by the SELECT statement cannot be updated after a fetch

ESQL

SPL

SELECT Clause

The SELECT clause contains the list of database objects or expressions to be selected, as shown in the following diagram.



Element	Purpose	Restrictions	Syntax
*	The asterisk (*) signifies that all columns in the specified table or view are to be selected.	Use this symbol whenever you want to retrieve all the columns in the table or view in their defined order. If you want to retrieve all the columns in some other order, or if you want to retrieve a subset of the columns, you must specify the columns explicitly in the SELECT list.	The asterisk (*) is a literal value that has a special meaning in this statement.
display label	A temporary name that you assign to a column. In DB-Access, the display label appears as the heading for the column in the output of the SELECT statement. In ESQL, the value of *display label* is stored in the **sqlname** field of the **sqlda** structure. For more information on the *display label* parameter, see "Using a Display Label" on page 1-466.	You can assign a display label to any column in your select list. If you are creating a temporary table with the SELECT...INTO TEMP clause, you must supply a display label for any columns that are not simple column expressions. The display label is used as the name of the column in the temporary table. If you are using the SELECT statement in creating a view, do not use display labels. Specify the desired label names in the CREATE VIEW column list instead. If your display label is also a keyword, you can use the AS keyword with the display label to clarify the use of the word. You must use the AS keyword with the display label to use any of the following words as a display label: UNITS, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION.	Identifier, p. 1-723

In the SELECT clause, specify exactly what data is being selected as well as whether you want to omit duplicate values.

Allowing Duplicates

You can apply the ALL, UNIQUE, or DISTINCT keywords to indicate whether duplicate values are returned, if any exist. If you do not specify any keywords, all the rows are returned by default.

Keyword	Meaning
ALL	Specifies that all selected values are returned, regardless of whether duplicates exist. ALL is the default state.
DISTINCT	Eliminates duplicate rows from the query results
UNIQUE	Eliminates duplicate rows from the query results. UNIQUE is a synonym for DISTINCT.

For example, the following query lists the **stock_num** and **manu_code** of all items that have been ordered, excluding duplicate items:

```
SELECT DISTINCT stock_num, manu_code FROM items
```

You can use the DISTINCT or UNIQUE keywords once in each level of a query or subquery. For example, the following query uses DISTINCT in both the query and the subquery:

```
SELECT DISTINCT stock_num, manu_code FROM items
    WHERE order_num = (SELECT DISTINCT order_num FROM orders
        WHERE customer_num = 120)
```

Expressions in the Select List

You can use any basic type of expression (column, constant, function, aggregate function, and stored procedure), or combination thereof, in the select list. The expression types are described in "Expression" on page 1-671.

The following sections present examples of using each type of simple expression in the select list.

You can combine simple numeric expressions by connecting them with arithmetic operators for addition, subtraction, multiplication, and division. However, if you combine a column expression and an aggregate function, you must include the column expression in the GROUP BY clause.

You cannot use variable names (for example, host variables in an ESQL application or stored procedure variables in a stored procedure) in the select list by themselves. You can include a variable name in the select list, however, if an arithmetic or concatenation operator connects it to a constant.

Selecting Columns

Column expressions are the most commonly used expressions in a SELECT statement. See "Column Expressions" on page 1-673 for a complete description of the syntax and use of column expressions.

The following examples show column expressions within a select list:

```
SELECT orders.order_num, items.price FROM orders, items

SELECT customer.customer_num ccnum, company FROM customer

SELECT catalog_num, stock_num, cat_advert [1,15] FROM catalog

SELECT lead_time - 2 UNITS DAY FROM manufact
```

Selecting Constants

If you include a constant expression in the select list, the same value is returned for each row that the query returns. See "Constant Expressions" on page 1-676 for a complete description of the syntax and use of constant expressions.

The following examples show constant expressions within a select list:

```
SELECT 'The first name is', fname FROM customer

SELECT TODAY FROM cust_calls

SELECT SITENAME FROM systables WHERE tabid = 1

SELECT lead_time - 2 UNITS DAY FROM manufact

SELECT customer_num + LENGTH('string') from customer
```

Selecting Function Expressions

A function expression uses a function that is evaluated for each row in the query. All function expressions require arguments. This set of expressions contains the time functions and the length function when they are used with a column name as an argument.

The following examples show function expressions within a select list:

```
SELECT EXTEND(res_dtime, YEAR TO SECOND) FROM cust_calls
SELECT LENGTH(fname) + LENGTH(lname) FROM customer
SELECT HEX(order_num) FROM orders
SELECT MONTH(order_date) FROM orders
```

Selecting Aggregate Expressions

Selecting Expressions That Use Arithmetic Operators

You can combine numeric expressions with arithmetic operators to make complex expressions. You cannot combine expressions that contain aggregate functions with column expressions. The following examples show expressions that use arithmetic operators within a select list:

```
SELECT stock_num, quantity*total_price FROM customer

SELECT price*2 doubleprice FROM items

SELECT count(*)+2 FROM customer

SELECT count(*)+LENGTH('ab') FROM customer
```

Using a Display Label

If you are creating a temporary table, you must supply a display label for any columns that are not simple column expressions. The display label is used as the name of the column in the temporary table.

DB

A display label appears as the heading for that column in the output of the SELECT statement. ♦

ESQL

The value of *display label* is stored in the **sqlname** field of the **sqlda** structure. See your SQL API product manual for more information on the **sqlda** structure. ♦

If you are using the SELECT statement in creating a view, do not use display labels. Specify the desired label names in the CREATE VIEW column list instead.

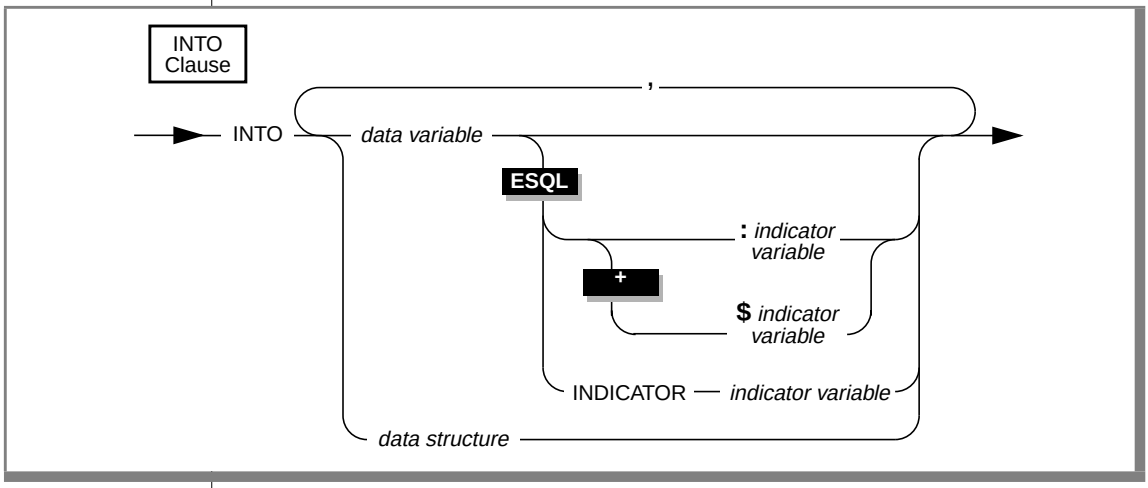
Using the AS Keyword

If your display label is also a keyword, you can use the AS keyword with the display label to clarify the use of the word. If you want to use the word UNITS, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION as your display label, you must use the AS keyword with the display label. The following example shows how to use the AS keyword with **minute** as a display label:

```
SELECT call_dtime AS minute FROM cust_calls
```

INTO Clause

Use the INTO clause within a stored procedure or SQL API to specify the program variables or host variables to receive the data that the SELECT statement retrieves. The following diagram shows the syntax of the INTO clause.



Element	Purpose	Restrictions	Syntax
data variable	A program variable or host object. This variable receives the value of the corresponding item in the select list of the SELECT clause	The order of receiving variables in the INTO clause must match the order of the corresponding items in the select list of the SELECT clause. The number of receiving variables must be equal to the number of items in the select list. The data type of each receiving variable should agree with the data type of the corresponding column or expression in the select list. See "Error Checking" on page 1-471 for the actions that the database server takes when the data type of the receiving variable does not match that of the selected item.	The name of the receiving variable must conform to language-specific rules for variable names.
data structure	A structure that has been declared as a host variable	The individual elements of the structure must be matched appropriately to the data type of values being selected.	The name of the data structure must conform to language-specific rules for data structures.
indicator variable	A program variable that receives a return code if null data is placed in the corresponding *data variable*	This parameter is optional, but you should use an indicator variable if the possibility exists that the value of the corresponding *data variable* is null.	The name of the indicator variable must conform to language-specific rules for indicator variables.

If the SELECT statement stands alone (that is, it is not part of a DECLARE statement and does not use the INTO clause), it must be a singleton SELECT statement. A singleton SELECT statement returns only one row. The following example shows a singleton SELECT statement in INFORMIX-ESQL / C:

```
EXEC SQL select fname, lname, company_name
    into :p_fname, :p_lname, :p_coname
    where customer_num = 101;
```

ESQL

INTO Clause with Indicator Variables

You should use an indicator variable if the possibility exists that data returned from the SELECT statement is null. See your SQL API product manual for more information about indicator variables.

INTO Clause with Cursors

If the SELECT statement returns more than one row, you must use a cursor in a FETCH statement to fetch the rows individually. You can put the INTO clause in the FETCH statement rather than in the SELECT statement, but you cannot put it in both.

The following INFORMIX-ESQL/C code examples show different ways you can use the INTO clause:

Using the INTO clause in the SELECT statement

```
EXEC SQL declare q_curs cursor for
    select lname, company
        into :p_lname, :p_company
        from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
    EXEC SQL fetch q_curs;
EXEC SQL close q_curs;
```

Using the INTO clause in the FETCH statement

```
EXEC SQL declare q_curs cursor for
    select lname, company
    from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
    EXEC SQL fetch q_curs into :p_lname, :p_company;
EXEC SQL close q_curs;
```

Preparing a SELECT...INTO Query

You cannot prepare a query that has an INTO clause. You can prepare the query without the INTO clause, declare a cursor for the prepared query, open the cursor, and then use the FETCH statement with an INTO clause to fetch the cursor into the program variable. Alternatively, you can declare a cursor for the query without first preparing the query and include the INTO clause in the query when you declare the cursor. Then open the cursor, and fetch the cursor without using the INTO clause of the FETCH statement.

Using Array Variables with the INTO Clause

If you use a DECLARE statement with a SELECT statement that contains an INTO clause, and the program variable is an array element, you can identify individual elements of the array with integer constants or with variables. The value of the variable that is used as a subscript is determined when the cursor is declared, so afterward the subscript variable acts as a constant.

The following INFORMIX-ESQL/C code example declares a cursor for a SELECT...INTO statement using the variables **i** and **j** as subscripts for the array **a**. After you declare the cursor, the INTO clause of the SELECT statement is equivalent to `INTO a[5], a[2]`.

```
i = 5
j = 2
EXEC SQL declare c cursor for
    select order_num, po_num into :a[i], :a[j] from orders
        where order_num =1005 and po_num =2865
```

You can also use program variables in the FETCH statement to specify an element of a program array in the INTO clause. With the FETCH statement, the program variables are evaluated at each fetch rather than when you declare the cursor. ♦

Error Checking

ESQL

If the number of variables that are listed in the INTO clause differs from the number of items in the SELECT clause, a warning is returned in the **sqlwarn** structure; the following diagram shows the specific structure name. The actual number of variables that are transferred is the lesser of the two numbers. See your SQL API product manual for information about the **sqlwarn** structure.

Product Name	Variable Name
ESQL / C	sqlca.sqlwarn.sqlwarn3
ESQL / COBOL	SQLWARN3 OF SQLWARN OF SQLCA

♦

ANSI

If the number of variables that are listed in the INTO clause differs from the number of items in the SELECT clause, you receive an error. ♦

ESQL

SPL

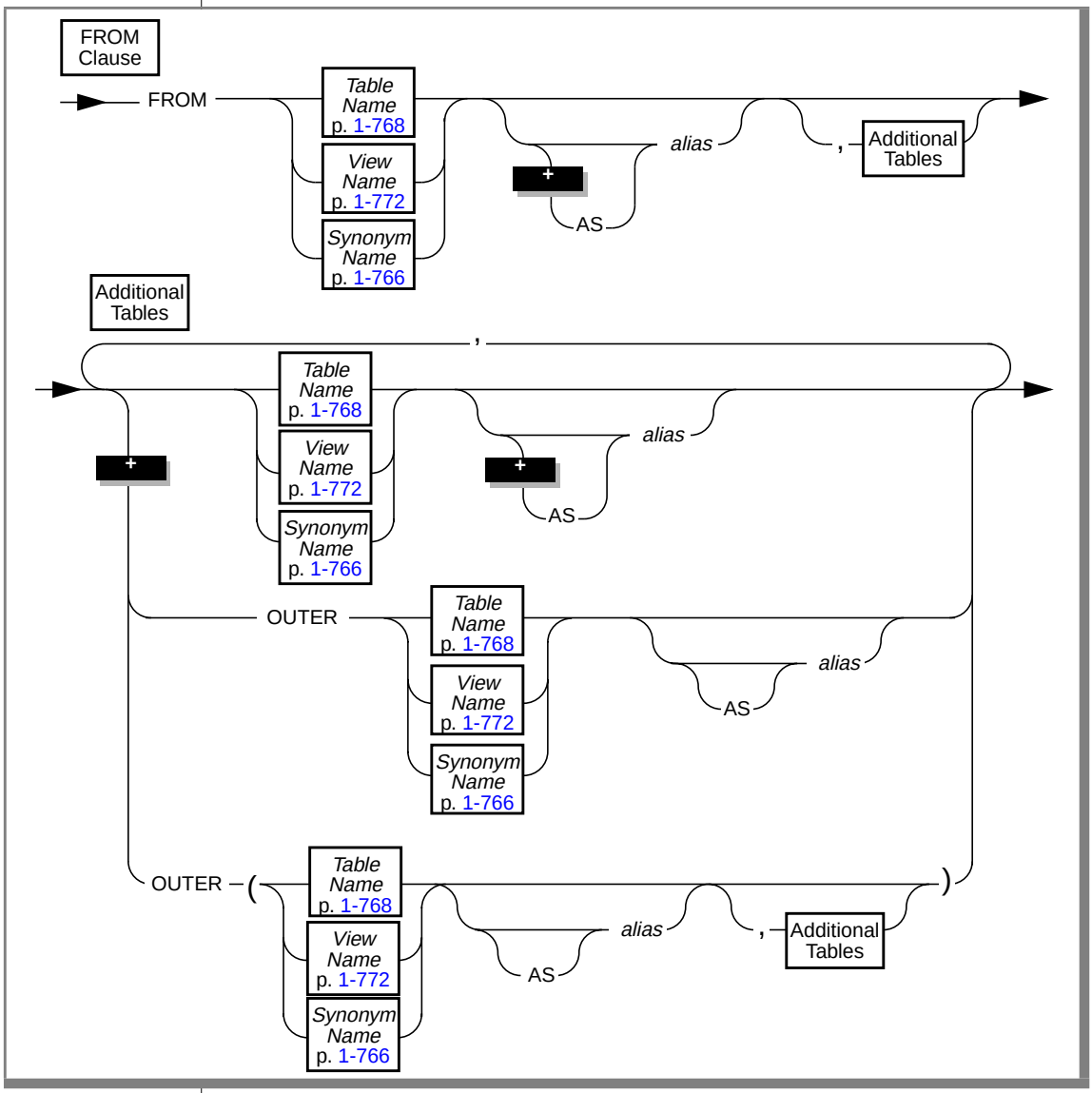
If the data type of the receiving variable does not match that of the selected item, the data type of the selected item is converted, if possible. If the conversion is impossible, an error occurs, and a negative value is returned in the status variable. In this case, the value in the program variable is unpredictable. The following table shows the specific name of the status variable for each application development tool.

Product Name	Variable Name
ESQL / C	sqlca.sqlcode, SQLCODE
ESQL / COBOL	SQLCODE OF SQLCA

♦

FROM Clause

The FROM clause lists the table or tables from which you are selecting the data. The following diagram shows the syntax of the FROM clause.



Element	Purpose	Restrictions	Syntax
alias	A temporary alternative name for a table or view within the scope of a SELECT statement. You can use aliases to make a query shorter.	If the SELECT statement is a self-join, you must list the table name twice in the FROM clause and assign a different alias to each occurrence of the table name. If you use a potentially ambiguous word as an alias, you must precede the alias with the keyword AS. See "AS Keyword with Aliases" on page 1-474 for further information on this restriction.	Identifier, p. 1-723

Use the keyword OUTER to form outer joins. Outer joins preserve rows that otherwise would be discarded by simple joins. See Chapter 3 of the *Informix Guide to SQL: Tutorial* for more information on outer joins.

You can supply an alias for a table name or view name. You can use the alias to refer to the table or view in other clauses of the SELECT statement. This is especially useful with a self-join. (See the WHERE clause on page 1-474 for more information about self-joins.)

The following example shows typical uses of the FROM clause. The first query selects all the columns and rows from the **customer** table. The second query uses a join between the **customer** and **orders** table to select all the customers who have placed orders.

```
SELECT * FROM customer

SELECT fname, lname, order_num
    FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
```

The following example is the same as the second query in the preceding example, except that it establishes aliases for the tables in the FROM clause and uses them in the WHERE clause:

```
SELECT fname, lname, order_num
    FROM customer c, orders o
    WHERE c.customer_num = o.customer_num
```

The following example uses the OUTER keyword to create an outer join and produce a list of all customers and their orders, regardless of whether they have placed orders:

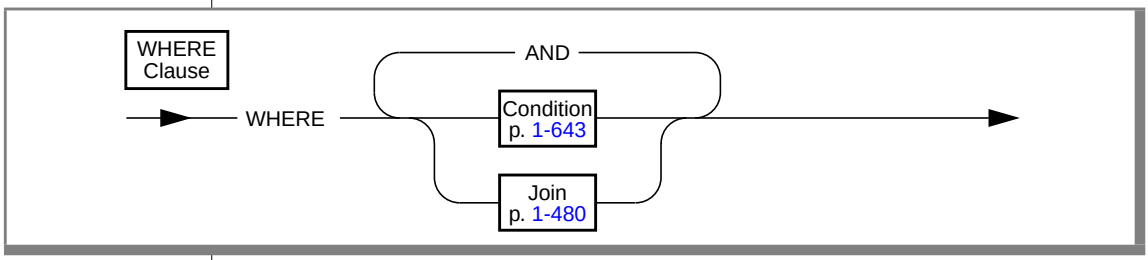
```
SELECT customer.customer_num, lname, order_num
    FROM customer c, OUTER orders o
    WHERE c.customer_num = o.customer_num
```

AS Keyword with Aliases

To use potentially ambiguous words as an alias for a table or view, you must precede them with the keyword AS. Use the AS keyword if you want to use the words ORDER, FOR, AT, GROUP, HAVING, INTO, UNION, WHERE, WITH, CREATE, or GRANT as an alias for a table or view.

WHERE Clause

Use the WHERE clause to specify search criteria and join conditions on the data that you are selecting.



Using a Condition in the WHERE Clause

You can use the following kinds of simple conditions or comparisons in the WHERE clause:

- Relational-operator condition
- BETWEEN
- IN
- IS NULL
- LIKE or MATCHES

You also can use a SELECT statement within the WHERE clause; this is called a subquery. The following list contains the kinds of subquery WHERE clauses:

- IN
- EXISTS
- ALL / ANY / SOME

Examples of each type of condition are shown in the following sections. For more information about each kind of condition, see the Condition segment on page 1-643.

You cannot use an aggregate function in the WHERE clause unless it is part of a subquery or if the aggregate is on a correlated column originating from a parent query and the WHERE clause is within a subquery that is within a HAVING clause.

Relational-Operator Condition

For a complete description of the relational-operator condition, see page 1-647.

A relational-operator condition is satisfied when the expressions on either side of the relational operator fulfill the relation that the operator set up. The following SELECT statements use the greater than (>) and equal (=) relational operators:

```
SELECT order_num FROM orders
    WHERE order_date > '6/04/94'

SELECT fname, lname, company
    FROM customer
    WHERE city[1,3] = 'San'
```

BETWEEN Condition

For a complete description of the BETWEEN condition, see page 1-648.

The BETWEEN condition is satisfied when the value to the left of the BETWEEN keyword lies in the inclusive range of the two values on the right of the BETWEEN keyword. The first two queries in the following example use literal values after the BETWEEN keyword. The third query uses the CURRENT function and a literal interval. It looks for dates between the current day and seven days earlier.

```
SELECT stock_num, manu_code FROM stock
    WHERE unit_price BETWEEN 125.00 AND 200.00

SELECT DISTINCT customer_num, stock_num, manu_code
    FROM orders, items
    WHERE order_date BETWEEN '6/1/93' AND '9/1/93'

SELECT * FROM cust_calls WHERE call_dtime
    BETWEEN (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT
```

IN Condition

For a complete description of the IN condition, see page 1-653.

The IN condition is satisfied when the expression to the left of the IN keyword is included in the list of values to the right of the keyword. The following examples show the IN condition:

```
SELECT lname, fname, company
    FROM customer
    WHERE state IN ('CA','WA', 'NJ')

SELECT * FROM cust_calls
    WHERE user_id NOT IN (USER )
```

IS NULL Condition

For a complete description of the IS NULL condition, see page 1-649.

The IS NULL condition is satisfied if the column contains a null value. If you use the NOT option, the condition is satisfied when the column contains a value that is not null. The following example selects the order numbers and customer numbers for which the order has not been paid:

```
SELECT order_num, customer_num FROM orders
    WHERE paid_date IS NULL
```

LIKE or MATCHES Condition

For a complete description of the LIKE or MATCHES condition, see page 1-649.

The LIKE or MATCHES condition is satisfied when either of the following tests is true:

- The value of the column that precedes the LIKE or MATCHES keyword matches the pattern that the quoted string specifies. You can use wildcard characters in the string.
- The value of the column that precedes the LIKE or MATCHES keyword matches the pattern that is specified by the column that follows the LIKE or MATCHES keyword. The value of the column on the right serves as the matching pattern in the condition.

The following SELECT statement returns all rows in the **customer** table in which the **lname** column begins with the literal string `'Baxter'`. Because the string is a literal string, the condition is case sensitive.

```
SELECT * FROM customer WHERE lname LIKE 'Baxter%'
```

The following SELECT statement returns all rows in the **customer** table in which the value of the **lname** column matches the value of the **fname** column:

```
SELECT * FROM customer WHERE lname LIKE fname
```

The following examples use the LIKE condition with a wildcard. The first SELECT statement finds all stock items that are some kind of ball. The second SELECT statement finds all company names that contain a percent sign (%). The backslash (\) is used as the standard escape character for the wildcard percent sign (%). The third SELECT statement uses the ESCAPE option with the LIKE condition to retrieve rows from the **customer** table in which the **company** column includes a percent sign (%). The `z` is used as an escape character for the wildcard percent sign (%).

```
SELECT stock_num, manu_code FROM stock
    WHERE description LIKE '%ball'

SELECT * FROM customer
    WHERE company LIKE '%\%%'

SELECT * FROM customer
    WHERE company LIKE '%z%%' ESCAPE 'z'
```

The following examples use MATCHES with a wildcard in several SELECT statements. The first SELECT statement finds all stock items that are some kind of ball. The second SELECT statement finds all company names that contain an asterisk (*). The backslash(\) is used as the standard escape character for the wildcard asterisk (*). The third statement uses the ESCAPE option with the MATCHES condition to retrieve rows from the **customer** table where the **company** column includes an asterisk (*). The z character is used as an escape character for the wildcard asterisk (*).

```
SELECT stock_num, manu_code FROM stock
```

The following example of a SELECT statement with an EXISTS subquery returns the stock number and manufacturer code for every item that has never been ordered (and is therefore not listed in the **items** table). It is appropriate to use an EXISTS subquery in this SELECT statement because you need the correlated subquery to test both **stock_num** and **manu_code** in the **items** table.

```
SELECT stock_num, manu_code FROM stock
    WHERE NOT EXISTS
        (SELECT stock_num, manu_code FROM items
            WHERE stock.stock_num = items.stock_num AND
                stock.manu_code = items.manu_code)
```

The preceding example would work equally well if you use a SELECT * in the subquery in place of the column names because you are testing for the existence of a row or rows.

ALL/ANY/SOME Subquery

For a complete description of the ALL / ANY / SOME subquery, see page 1-655.

In the following example, the SELECT statements return the order number of all orders that contain an item whose total price is greater than the total price of every item in order number 1023. The first SELECT statement uses the ALL subquery, and the second SELECT statement produces the same result by using the MAX aggregate function.

```
SELECT DISTINCT order_num FROM items
    WHERE total_price > ALL (SELECT total_price FROM items
         WHERE order_num = 1023)

SELECT DISTINCT order_num FROM items
    WHERE total_price > SELECT MAX(total_price) FROM items
         WHERE order_num = 1023)
```

The following SELECT statements return the order number of all orders that contain an item whose total price is greater than the total price of at least one of the items in order number 1023. The first SELECT statement uses the ANY keyword, and the second SELECT statement uses the MIN aggregate function.

```
SELECT DISTINCT order_num FROM items
    WHERE total_price > ANY (SELECT total_price FROM items
         WHERE order_num = 1023)

SELECT DISTINCT order_num FROM items
    WHERE total_price > (SELECT MIN(total_price) FROM items
        WHERE order_num = 1023)
```

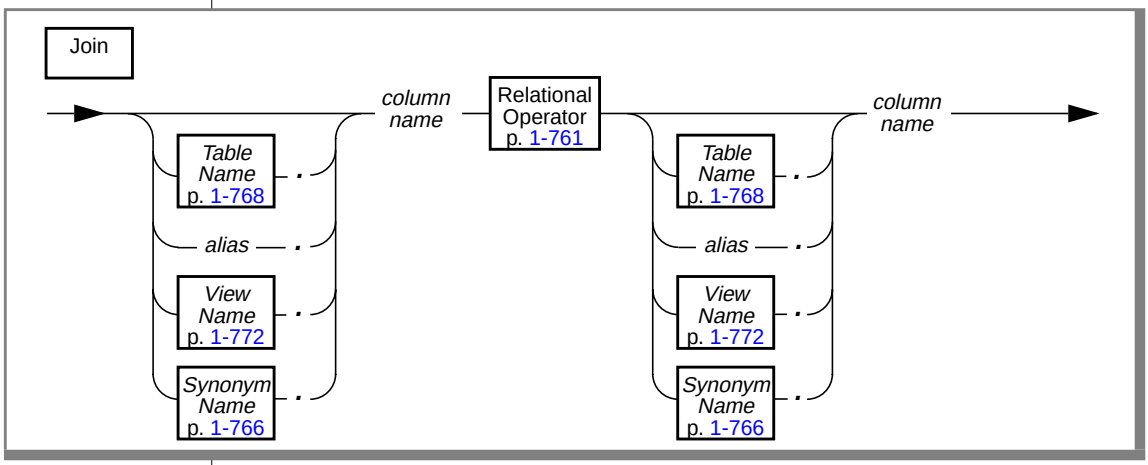
You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery returns exactly one value. If you omit ANY, ALL, or SOME, and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate function:

```
SELECT order_num FROM items
    WHERE stock_num = 9 AND quantity =
        (SELECT MAX(quantity) FROM items WHERE stock_num = 9)
```

Using a Join in the WHERE Clause

You join two tables when you create a relationship in the WHERE clause between at least one column from one table and at least one column from another table. The effect of the join is to create a temporary composite table where each pair of rows (one from each table) that satisfies the join condition is linked to form a single row. You can create two-table joins, multiple-table joins, and self-joins.

The following diagram shows the syntax for a join.



Element	Purpose	Restrictions	Syntax
alias	The alias assigned to the table or view in the FROM clause. See "FROM Clause" on page 1-472 for more information on aliases for tables and views.	If the tables to be joined are the same table (that is, if the join is a self-join), you must refer to each instance of the table in the WHERE clause by the alias assigned to that table instance in the FROM clause.	Identifier, p. 1-723
column name	The name of a column from one of the tables or views to be joined. Rows from the tables or views are joined when there is a match between the values of the specified columns.	When the specified columns have the same name in the tables or views to be joined, you must distinguish the columns by preceding each column name with the name or alias of the table or view in which the column resides.	Identifier, p. 1-723

Two-Table Joins

The following example shows a two-table join:

```
SELECT order_num, lname, fname
    FROM customer, orders
    WHERE customer.customer_num = orders.customer_num
```

Tip: *You do not have to select the column where the two tables are joined.*

Multiple-Table Joins

A multiple-table join is a join of more than two tables. Its structure is similar to the structure of a two-table join, except that you have a join condition for more than one pair of tables in the WHERE clause. When columns from different tables have the same name, you must distinguish them by preceding the name with its associated table or table alias, as in *table.column*. See "Table Name" on page 1-768 for the full syntax of a table name.

The following multiple-table join yields the company name of the customer who ordered an item as well as the stock number and manufacturer code of the item:

```
SELECT DISTINCT company, stock_num, manu_code
    FROM customer c, orders o, items i
    WHERE c.customer_num = o.customer_num
        AND o.order_num = i.order_num
```

Self-Joins

You can join a table to itself. To do so, you must list the table name twice in the FROM clause and assign it two different table aliases. Use the aliases to refer to each of the "two" tables in the WHERE clause.

The following example is a self-join on the **stock** table. It finds pairs of stock items whose unit prices differ by a factor greater than 2.5. The letters x and y are each aliases for the **stock** table.

```
SELECT x.stock_num, x.manu_code, y.stock_num, y.manu_code
    FROM stock x, stock y
    WHERE x.unit_price > 2.5 * y.unit_price
```

Outer Joins

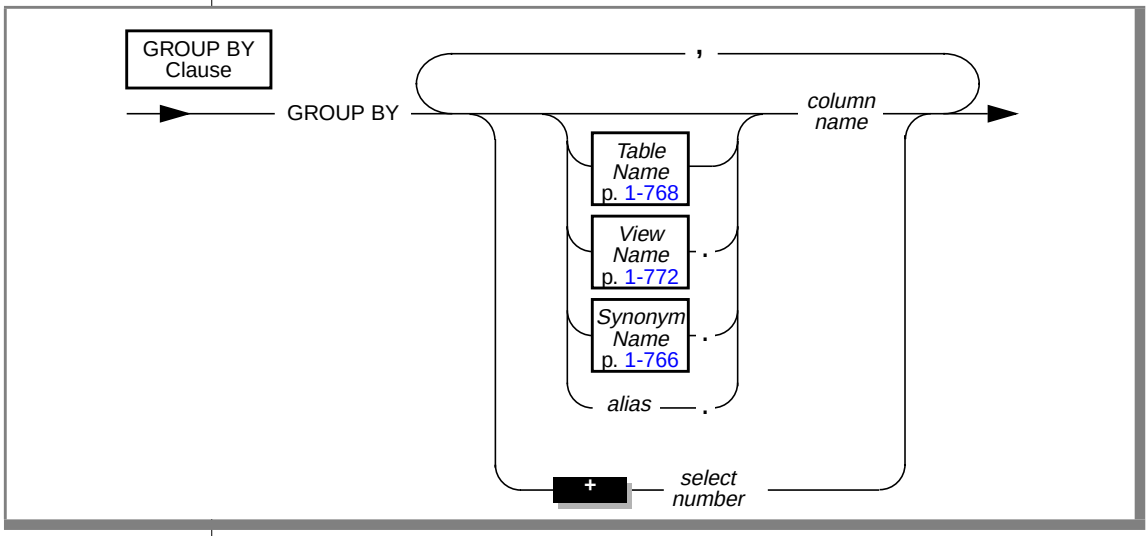
The following outer join lists the company name of the customer and all associated order numbers, if the customer has placed an order. If not, the company name is still listed, and a null value is returned for the order number.

```
SELECT company, order_num
    FROM customer c, OUTER orders o
    WHERE c.customer_num = o.customer_num
```

See Chapter 3 of the *Informix Guide to SQL: Tutorial* for more information about outer joins.

GROUP BY Clause

Use the GROUP BY clause to produce a single row of results for each group. A group is a set of rows that have the same values for each column listed.



Element	Purpose	Restrictions	Syntax
alias	The alias assigned to a table or view in the FROM clause. See "FROM Clause" on page 1-472 for more information on aliases for tables and views.	You cannot use an alias for a table or view in the GROUP BY clause unless you have assigned the alias to the table or view in the FROM clause.	Identifier, p. 1-723
column name	The name of a stand-alone column in the select list of the SELECT clause or the name of one of the columns joined by an arithmetic operator in the select list. The SELECT statement returns a single row of results for each group of rows that have the same value in *column name*.	See "Relationship of the GROUP BY Clause to the SELECT Clause" on page 1-484.	Identifier, p. 1-723

(1 of 2)

Element	Purpose	Restrictions	Syntax
select number	An integer that identifies a column or expression in the select list of the SELECT clause by specifying its order in the select list. The SELECT statement returns a single row of results for each group of rows that have the same value in the column or expression identified by *select number*.	See "Using Select Numbers" on page 1-485.	Literal Number, p. 1-752

(2 of 2)

Relationship of the GROUP BY Clause to the SELECT Clause

A GROUP BY clause restricts what you can enter in the SELECT clause. If you use a GROUP BY clause, each column that you select must be in the GROUP BY list. If you use an aggregate function and one or more column expressions in the select list, you must put all the column names that are not used as part of an aggregate or time expression in the GROUP BY clause. Do not put constant expressions or BYTE or TEXT column expressions in the GROUP BY list. If you are selecting a BYTE or TEXT column, you cannot use the GROUP BY clause. In addition, you cannot use ROWID in a GROUP BY clause.

The following example names one column that is not in an aggregate expression. The **total_price** column should not be in the GROUP BY list because it appears as the argument of an aggregate function. The COUNT and SUM keywords are applied to each group, not the whole query set.

```
SELECT order_num, COUNT(*), SUM(total_price)
    FROM items
    GROUP BY order_num
```

If a column stands alone in a column expression in the select list, you must use it in the GROUP BY clause. If a column is combined with another column by an arithmetic operator, you can choose to group by the individual columns or by the combined expression using a specific number.

Using Select Numbers

You can use one or more integers in the GROUP BY clause to stand for column expressions. In the following example, the first SELECT statement uses select numbers for **order_date** and **paid_date - order_date** in the GROUP BY clause. Note that you can group only by a combined expression using the select-number notation. In the second SELECT statement, you cannot replace the 2 with the expression **paid_date - order_date**.

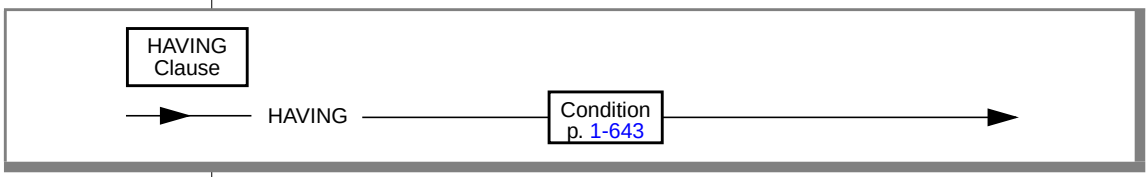
```
SELECT order_date, COUNT(*), paid_date - order_date
    FROM orders
    GROUP BY 1, 3

SELECT order_date, paid_date - order_date
    FROM orders
    GROUP BY order_date, 2
```

Nulls in the GROUP BY Clause

Each row that contains a null value in a column that is specified by a GROUP BY clause belongs to a single group (that is, all null values are grouped together).

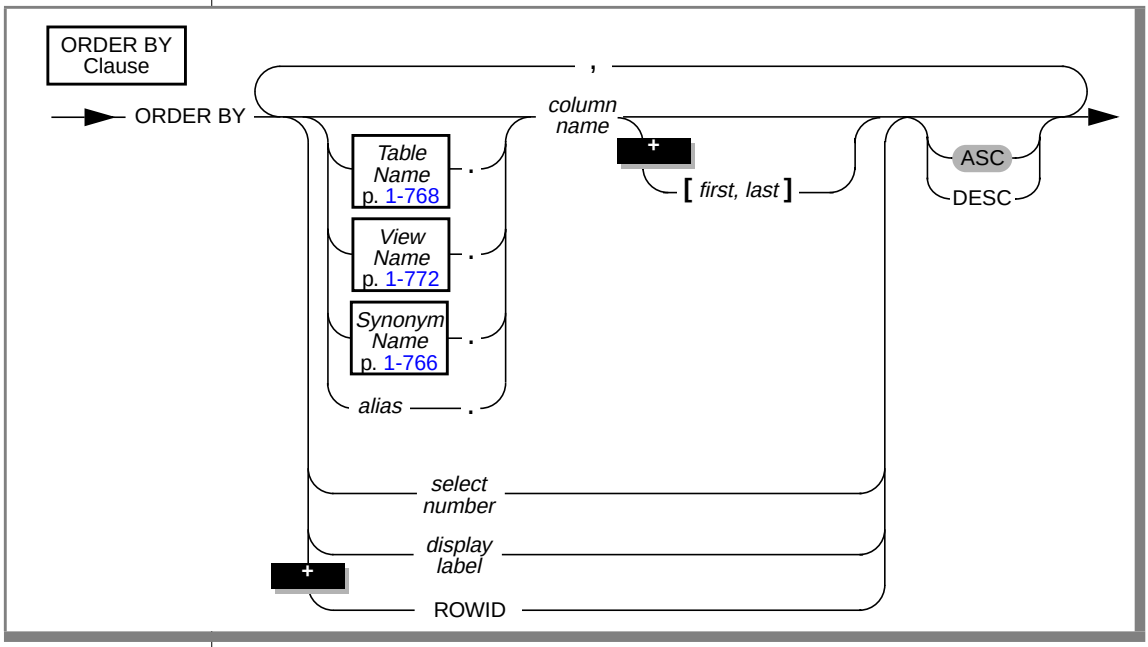
HAVING Clause

Use the HAVING clause to apply one or more qualifying conditions to groups.



ORDER BY Clause

Use the ORDER BY clause to sort query results by the values that are contained in one or more columns.



Element	Purpose	Restrictions	Syntax
alias	The alias assigned to a table or view in the FROM clause. See "FROM Clause" on page 1-472 for more information on aliases for tables and views.	You cannot specify an alias for a table or view in the ORDER BY clause unless you have assigned the alias to the table or view in the FROM clause.	Identifier, p. 1-723
column name	The name of a column in the specified table or view. The query results are sorted by the values contained in this column.	A column specified in the ORDER BY clause must be listed explicitly or implicitly in the select list of the SELECT clause. If you want to order the query results by a derived column, you must supply a display label for the derived column in the select list and specify this label in the ORDER BY clause. Alternatively, you can omit a display label for the derived column in the select list and specify the derived column by means of a select number in the ORDER BY clause.	Identifier, p. 1-723
display label	A temporary name that you assign to a column in the select list of the SELECT clause. You can use a display label in place of the column name in the ORDER BY clause.	You cannot specify a display label in the ORDER BY clause unless you have specified this display label for a column in the select list.	Identifier, p. 1-723
first	The position of the first character in the portion of the column that is used to sort the query results	The column must be one of the following character types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-752
last	The position of the last character in the portion of the column that is used to sort the query results	The column must be one of the following character types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-752
select number	An integer that identifies a column in the select list of the SELECT clause by specifying its order in the select list. You can use a select number in place of a column name in the ORDER BY clause.	You must specify select numbers in the ORDER BY clause when SELECT statements are joined by UNION or UNION ALL keywords and compatible columns in the same position have different names.	Literal Number, p. 1-752

You can perform an ORDER BY operation on a column or on an aggregate expression when you use SELECT * or a display label in your SELECT statement.

The following query explicitly selects the order date and shipping date from the **orders** table and then rearranges the query by the order date. By default, the query results are listed in ascending order.

```
SELECT order_date, ship_date FROM orders
    ORDER BY order_date
```

In the following query, the **order_date** column is selected implicitly by the SELECT * statement, so you can use **order_date** in the ORDER BY clause:

```
SELECT * FROM orders
    ORDER BY order_date
```

Ordering by a Column Substring

You can order by a column substring instead of ordering by the entire length of the column. The column substring is the portion of the column that the database server uses for the sort. You define the column substring by specifying column subscripts (the *first* and *last* parameters). The column subscripts represent the starting and ending character positions of the column substring.

The following example shows a SELECT statement that queries the **customer** table and specifies a column substring in the ORDER BY column. The column substring instructs the database server to sort the query results by the portion of the **lname** column contained in the sixth through ninth positions of the column:

```
SELECT * from customer
    ORDER BY lname[6,9]
```

Assume that the value of **lname** in one row of the **customer** table is `Greenburg`. Because of the column substring in the ORDER BY clause, the database server determines the sort position of this row by using the value `burg`, not the value `Greenburg`.

You can specify column substrings only for columns that have a character data type. If you specify a column substring in the ORDER BY clause, the column must have one of the following data types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.

GLS

For information on the GLS aspects of using column substrings in the ORDER BY clause, see the *Guide to GLS Functionality*, Chapter 3. ♦

Ordering by a Derived Column

You can order by a derived column by supplying a display label in the SELECT clause, as shown in the following example:

```
SELECT paid_date - ship_date span, customer_num
    FROM orders
    ORDER BY span
```

Ascending and Descending Orders

You can use the ASC and DESC keywords to specify ascending (smallest value first) or descending (largest value first) order. The default order is ascending.

For DATE and DATETIME data types, *smallest* means earliest in time and *largest* means latest in time. For standard character data types, the ASCII collating sequence is used. See page 1-763 for a listing of the collating sequence.

Nulls in the ORDER BY Clause

Null values are ordered as less than values that are not null. Using the ASC order, the null value comes before the non-null value; using DESC order, the null value comes last.

Nested Ordering

If you list more than one column in the ORDER BY clause, your query is ordered by a nested sort. The first level of sort is based on the first column; the second column determines the second level of sort. The following example of a nested sort selects all the rows in the **cust_calls** table and orders them by **call_code** and by **call_dtime** within **call_code**:

```
SELECT * FROM cust_calls
    ORDER BY call_code, call_dtime
```

Using Select Numbers

In place of column names, you can enter one or more integers that refer to the position of items in the SELECT clause. You can use a select number to order by an expression. For instance, the following example orders by the expression **paid_date - order_date** and **customer_num**, using select numbers in a nested sort:

```
SELECT order_num, customer_num, paid_date - order_date
    FROM orders
    ORDER BY 3, 2
```

Select numbers are required in the ORDER BY clause when SELECT statements are joined by the UNION or UNION ALL keywords and compatible columns in the same position have different names.

Ordering by Rowids

You can specify the **rowid** column as a column in the ORDER BY clause. The **rowid** column is a hidden column in nonfragmented tables and in fragmented tables that were created with the WITH ROWIDS clause. The **rowid** column contains a unique internal record number that is associated with a row in a table. Informix recommends, however, that you utilize primary keys as an access method rather than exploiting the **rowid** column.

If you want to specify the **rowid** column in the ORDER BY clause, enter the keyword ROWID in lowercase or uppercase letters.

You cannot specify the **rowid** column in the ORDER BY clause if the table you are selecting from is a fragmented table that does not have a rowid column.

You cannot specify the **rowid** column in the ORDER BY clause unless you have included the **rowid** column in the select list of the SELECT clause.

For further information on using the **rowid** column in column expressions, see "Expression" on page 1-671.

ORDER BY Clause with DECLARE

ESQL

You cannot use a DECLARE statement with a FOR UPDATE clause to associate a cursor with a SELECT statement that has an ORDER BY clause. ♦

Placing Indexes on ORDER BY Columns

When you include an ORDER BY clause in a SELECT statement, you can improve the performance of the query by creating an index on the column or columns that the ORDER BY clause specifies. The database server uses the index that you placed on the ORDER BY columns to sort the query results in the most efficient manner. For further information on creating indexes that correspond to the columns of an ORDER BY clause, see "The ASC and DESC Keywords" on page 1-113 under the CREATE INDEX statement.

FOR UPDATE Clause

Use the FOR UPDATE clause when you prepare a SELECT statement, and you intend to update the values returned by the SELECT statement when the values are fetched. Preparing a SELECT statement that contains a FOR UPDATE clause is equivalent to preparing the SELECT statement without the FOR UPDATE clause and then declaring a FOR UPDATE cursor for the prepared statement.

The FOR UPDATE keyword notifies the database server that updating is possible, causing it to use more-stringent locking than it would with a select cursor. You cannot modify data through a cursor without this clause. You can specify particular columns that can be updated.

After you declare a cursor for a SELECT... FOR UPDATE statement, you can update or delete the currently selected row using an UPDATE or DELETE statement with the WHERE CURRENT OF clause. The words CURRENT OF refer to the row that was most recently fetched; they replace the usual test expressions in the WHERE clause.

To update rows with a particular value, your program might contain statements such as the sequence of statements shown in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
    char fname[ 16];
    char lname[ 16];
    EXEC SQL END DECLARE SECTION;
.
.
.

 EXEC SQL connect to 'stores7';
 /* select statement being prepared contains a for update clause */
 EXEC SQL prepare x from 'select fname, lname from customer for update';
 EXEC SQL declare xc cursor for x; --note no 'for update' clause in declare
```

```
for (;;)
   {
   EXEC SQL fetch xc into $fname, $lname;
   if (strncmp(SQLSTATE, '00', 2) != 0) break;
   printf("%d %s %s\n",cnum, fname, lname );
   if (cnum == 999)--update rows with 999 customer_num
       EXEC SQL update customer set fname = 'rosey' where current of xc;
   }

EXEC SQL close xc;
EXEC SQL disconnect current;
```

A SELECT ... FOR UPDATE statement, like an update cursor, allows you to perform updates that are not possible with the UPDATE statement alone, because both the decision to update and the values of the new data items can be based on the original contents of the row. The UPDATE statement cannot interrogate the table that is being updated.

Syntax That is Incompatible with the FOR UPDATE Clause

A SELECT statement that uses a FOR UPDATE clause must conform to the following restrictions:

- The statement can select data from only one table.
- The statement cannot include any aggregate functions.
- The statement cannot include any of the following clauses or keywords: DISTINCT, FOR READ ONLY, GROUP BY, INTO TEMP, ORDER BY, UNION, or UNIQUE.

For information on how to declare an update cursor for a SELECT statement that does not include a FOR UPDATE clause, see page 1-242.

FOR READ ONLY Clause

Use the FOR READ ONLY clause to specify that the select cursor declared for the SELECT statement is a read-only cursor. A read-only cursor is a cursor that cannot modify data. This section provides the following information about the FOR READ ONLY clause:

- When you must use the FOR READ ONLY clause
- Syntax restrictions on a SELECT statement that uses a FOR READ ONLY clause

Using the FOR READ ONLY Clause in Read-Only Mode

Normally, you do not need to include the FOR READ ONLY clause in a SELECT statement. A SELECT statement is a read-only operation by definition, so the FOR READ ONLY clause is usually unnecessary. However, in certain special circumstances, you must include the FOR READ ONLY clause in a SELECT statement.

ANSI

If you have used the High-Performance Loader (HPL) in express mode to load data into the tables of an ANSI-mode database, and you have not yet performed a level-0 backup of this data, the database is in read-only mode. When the database is in read-only mode, the database server rejects any attempts by a select cursor to access the data unless the SELECT or the DECLARE includes a FOR READ ONLY clause. This restriction remains in effect until the user has performed a level-0 backup of the data.

When the database is an ANSI-mode database, select cursors are update cursors by default. An update cursor is a cursor that can be used to modify data. These update cursors are incompatible with the read-only mode of the database. For example, the following SELECT statement against the **customer_ansi** table fails:

```
EXEC SQL declare ansi_curs cursor for
    select * from customer_ansi;
```

The solution is to include the FOR READ ONLY clause in your select cursors. The read-only cursor that this clause specifies is compatible with the read-only mode of the database. For example, the following SELECT FOR READ ONLY statement against the **customer_ansi** table succeeds:

```
EXEC SQL declare ansi_read cursor for
    select * from customer_ansi for read only;
```

♦

D/B

DB-Access executes all SELECT statements with select cursors. Therefore, you must include the FOR READ ONLY clause in all SELECT statements that access data in a read-only ANSI-mode database. The FOR READ ONLY clause causes DB-Access to declare the cursor for the SELECT statement as a read-only cursor. ♦

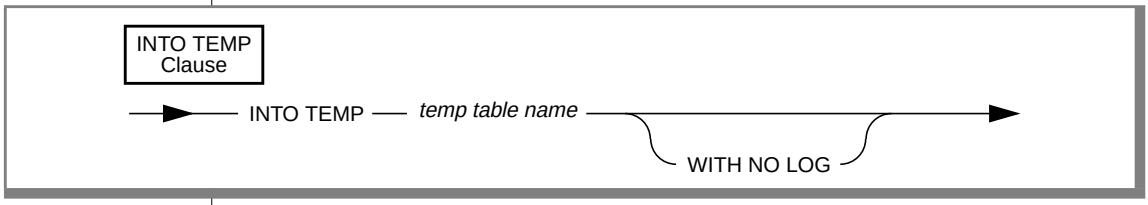
For more information on the express mode of HPL, see the *Guide to the High-Performance Loader*. For more information on level-0 backups, see the *INFORMIX-OnLine Dynamic Server Archive and Backup Guide*. For more information on select cursors, read-only cursors, and update cursors, see the DECLARE statement on page 1-234.

Syntax That Is Incompatible with the FOR READ ONLY Clause

Whether your database is an ANSI-mode database or a database that is not ANSI compliant, you cannot include both the FOR READ ONLY clause and the FOR UPDATE clause in the same SELECT statement. If you attempt to do so, the SELECT statement fails.

For information on how to declare a read-only cursor for a SELECT statement that does not include a FOR READ ONLY clause, see page 1-245.

INTO TEMP Clause



Element	Purpose	Restrictions	Syntax
temp table name	The simple name of a temporary table. This table contains the results of the SELECT statement. The column names of the temporary table are those that are named in the select list of the SELECT clause.	The name must be different from any existing table, view, or synonym name in the current database, but it does not have to be different from other temporary table names used by other users. You must have the Connect privilege on a database to create a temporary table in that database. If you use the INTO TEMP clause to create a temporary table, you must supply a display label for all expressions in the select list other than simple column expressions.	Identifier, p. 1-723

The INTO TEMP clause creates a temporary table that contains the query results. The initial and next extents for the temporary table are always eight pages. Temporary tables created with the INTO TEMP clause are *explicit* temporary tables. Explicit temporary tables can also be created with the CREATE TEMP TABLE statement.

If the **DBSPACETEMP** environment variable is set for INFORMIX-OnLine Dynamic Server, temporary tables created with the INTO TEMP clause are located in the dbspaces that are specified in the **DBSPACETEMP** list. You can also specify dbspace settings with the ONCONFIG parameter **DBSPACETEMP**. If neither the environment variable nor configuration parameter is set, the default setting is the root dbspace. The settings specified for the **DBSPACETEMP** environment variable take precedence over the ONCONFIG parameter **DBSPACETEMP** and the default setting. For more information about creating temporary tables, see the CREATE TABLE statement on page 1-154. For more information about the **DBSPACETEMP** environment variable, see Chapter 4 of the *Informix Guide to SQL: Reference*. For more information about the ONCONFIG parameter **DBSPACETEMP**, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

SE

Temporary tables are located in whatever directory is specified in the **DBTEMP** environment variable setting or in the directory of the database (that is, the *.**dbs** directory). ♦

The temporary table disappears when your program ends or when you issue a DROP TABLE statement on the temporary table. If your database does not have logging, or if it has logging, and you created the temporary table without the WITH NO LOG keywords, the temporary table disappears when you close the current database.

If you use the same query results more than once, using a temporary table saves time. In addition, using an INTO TEMP clause often gives you clearer and more understandable SELECT statements. However, the data in the temporary table is static; data is not updated as changes are made to the tables used to build the temporary table.

The column names of the temporary table are those named in the SELECT clause. You must supply a display label for all expressions other than simple column expressions. The display label for a column or expression becomes the column name in the temporary table. If you do not provide a display label for a column expression, the temporary table uses the column name from the select list. The following example creates the **pushdate** table with two columns, **customer_num** and **slowdate**:

```
SELECT customer_num, call_dtime + 5 UNITS DAY slowdate
    FROM cust_calls INTO TEMP pushdate
```

You can put indexes on a temporary table.

INTO TEMP Clause and WHERE Clause

When you use the INTO TEMP clause combined with the WHERE clause, and no rows are returned, the **SQLNOTFOUND** value is 100 in ANSI-compliant databases and 0 in databases that are not ANSI compliant. If the SELECT INTO TEMP ... WHERE ... statement is a part of a multistatement prepare and no rows are returned, the **SQLNOTFOUND** value is 100 for both ANSI-compliant databases and databases that are not ANSI-compliant.

ESQL

INTO TEMP Clause and INTO

Do not use the INTO option with the INTO TEMP clause: If you do, no results are returned to the program variables and the **sqlcode** variable is set to a negative value. The name of the **sqlcode** variable for each product is shown in the following table.

Product	Variable Name
ESQL / C	sqlca.sqlcode, SQLCODE
ESQL / COBOL	SQLCODE OF SQLCA

♦

WITH NO LOG Option

If you use the WITH NO LOG keywords, operations on the temporary table are not included in the transaction-log operations. You can use this option to reduce the overhead of transaction logging.

UNION Operator

Place the UNION operator between two SELECT statements to combine the queries into a single query. You can string several SELECT statements together using the UNION operator. Corresponding items do not need to have the same name.

Restrictions on a Combined SELECT

Several restrictions apply on the queries that you can connect with a UNION operator, as the following list describes:

- The number of items in the SELECT clause of each query must be the same, and the corresponding items in each SELECT clause must have compatible data types.
- If you use an ORDER BY clause, it must follow the last SELECT clause, and you must refer to the item ordered by integer, not by identifier. Ordering takes place after the set operation is complete.

ESQL

- You cannot use a UNION operator inside a subquery or in the definition of a view.
- You cannot use an INTO clause in a query unless you are sure that the compound query returns exactly one row, and you are not using a cursor. In this case, the INTO clause must be in the first SELECT statement. ♦

To put the results of a UNION operator into a temporary table, use an INTO TEMP clause in the final SELECT statement.

Duplicate Rows in a Combined SELECT

If you use the UNION operator alone, the duplicate rows are removed from the complete set of rows. That is, if multiple rows contain identical values in each column, only one row is retained. If you use the UNION ALL operator, all the selected rows are returned (the duplicates are not removed). The following example uses the UNION ALL operator to join two SELECT statements without removing duplicates. The query returns a list of all the calls that were received during the first quarter of 1993 and the first quarter of 1994.

```
SELECT customer_num, call_code FROM cust_calls
    WHERE call_dtime BETWEEN
        DATETIME (1993-1-1) YEAR TO DAY
        AND DATETIME (1993-3-31) YEAR TO DAY

UNION ALL

SELECT customer_num, call_code FROM cust_calls
    WHERE call_dtime BETWEEN
        DATETIME (1994-1-1)YEAR TO DAY
        AND DATETIME (1994-3-31) YEAR TO DAY
```

If you want to remove duplicates, use the UNION operator without the keyword ALL in the query. In the preceding example, if the combination `101 B` were returned in both SELECT statements, a UNION operator would cause the combination to be listed once. (If you want to remove duplicates within each SELECT statement, use the DISTINCT keyword in the SELECT clause, as described on page 1-461.)

References

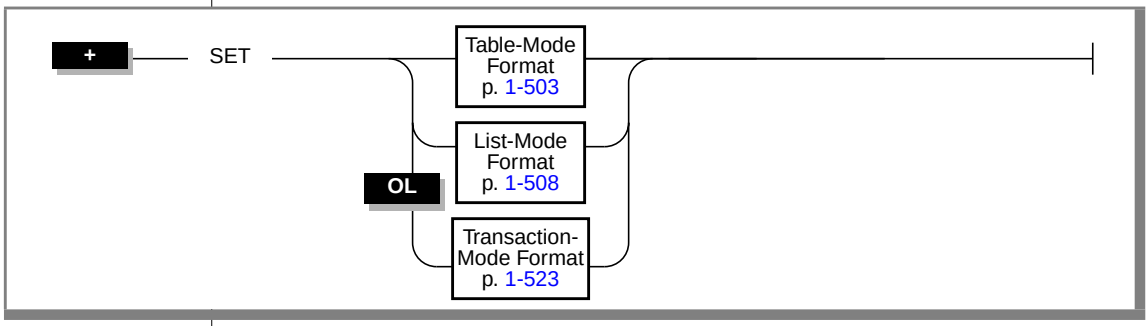
In the *Informix Guide to SQL: Tutorial*, see the discussion of the SELECT statement in Chapter 2 and Chapter 3.

In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the SELECT statement.

SET

The SET statement allows you to change the object mode of the following database objects: constraints, indexes, and triggers. You can also use the SET statement to specify the transaction mode of constraints.

Syntax



Usage

The SET statement has the following purposes:

- To change the object mode of constraints, indexes, and triggers

  When you change the object mode of constraints, indexes, or triggers, the change is permanent. The setting that the SET statement produces remains in effect until you change the object mode of the object again.

- To set the transaction mode of constraints by specifying whether constraints are checked at the statement level or at the transaction level

  When you set the transaction mode of constraints, the effect of the SET statement is limited to the transaction in which it is executed. The setting that the SET statement produces is effective only during the transaction. For further information on setting the transaction mode for constraints, see "Transaction-Mode Format" on page 1-523.

Terminology for Object Modes

The SET statement operates on database objects by changing the object mode of those objects. The terms *database objects* and *objects* have a restricted meaning in the context of the SET statement. Both terms refer to the constraints, indexes, and triggers in a database.

Similarly, the term *object modes* has a restricted meaning in the context of the SET statement. The term refers to the three states that a database object can have: enabled, disabled, and filtering. The **sysobjstate** system catalog table lists all of the objects in the database and the current object mode of each object.

Do not confuse the terms *objects* and *object modes* as used in the SET statement with the term *objects* in INFORMIX-NewEra. In the context of INFORMIX-NewEra, *objects* refers to objects within an application.

Methods for Changing Object Modes

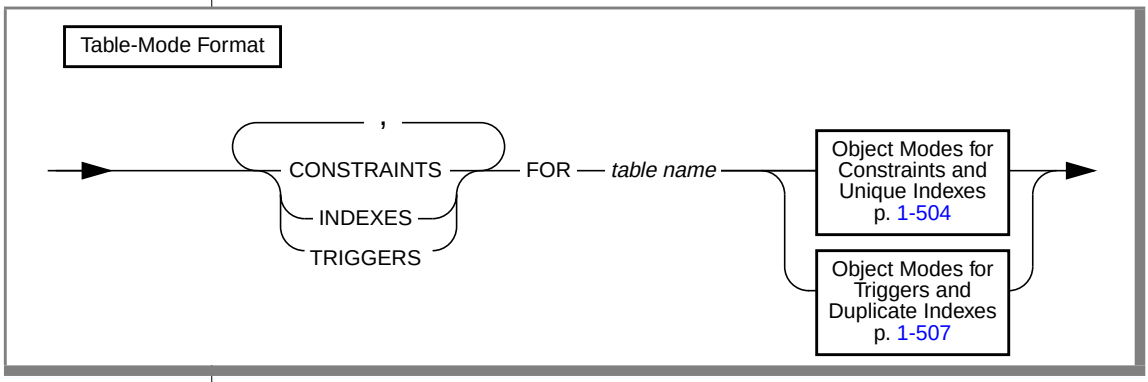
The SET statement provides the following formats for changing object modes: table mode and list mode. For an explanation of the table-mode format, see "Table-Mode Format" on page 1-503. For an explanation of the list-mode format, see "List-Mode Format" on page 1-508.

Privileges Required for Changing Object Modes

To change the object mode of a constraint, index, or trigger, you must have the necessary privileges. Specifically, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the table on which the object is defined and must have the Resource privilege on the database.
- You must have the Alter privilege on the table on which the object is defined and the Resource privilege on the database.

Table-Mode Format



Element	Purpose	Restrictions	Syntax
table name	The name of the table whose objects will have their object mode changed. There is no default value.	The table must be a local table. You cannot set the object modes of objects defined on a temporary table to the disabled or filtering modes. For information on the privileges required to change the object mode of the objects defined on a table, see "Privileges Required for Changing Object Modes" on page 1-502.	Identifier, p. 1-723

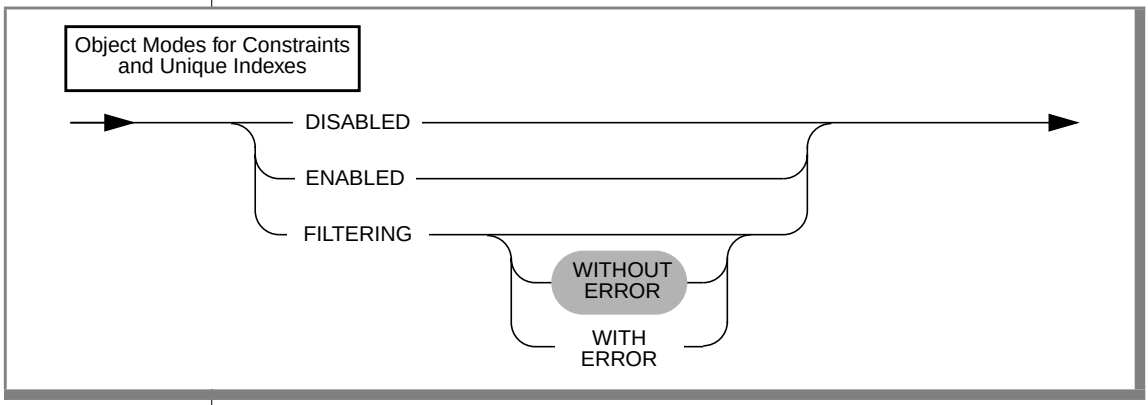
Use the table-mode format to change the object mode of all objects of a given type that have been defined on a particular table. For example, to change the object mode of all constraints that are defined on the **cust_subset** table to the disabled mode, enter the following statement:

```
SET CONSTRAINTS FOR cust_subset DISABLED
```

By using the table-mode format, you can change the object modes of more than one object type with a single SET statement. For example, to change the object mode of all constraints, indexes, and triggers that are defined on the **cust_subset** table to the enabled mode, enter the following statement:

```
SET CONSTRAINTS, INDEXES, TRIGGERS FOR cust_subset
    ENABLED
```

Object Modes for Constraints and Unique Indexes



You can specify the disabled, enabled, or filtering object modes for a constraint or a unique index. You must specify one of these object modes in your SET statement. The SET statement has no default object mode.

You can also specify the object mode for a constraint when you create the constraint with the ALTER TABLE or CREATE TABLE statements. If you do not specify the object mode for a constraint in one of these statements or in a SET statement, the constraint is in the enabled object mode by default.

You can also specify the object mode for a unique index when you create the index with the CREATE INDEX statement. If you do not specify the object mode for a unique index in the CREATE INDEX statement or in a SET statement, the unique index is in the enabled object mode by default.

For definitions of the disabled, enabled, and filtering object modes see "Using Object Modes with Data Manipulation Statements" on page 1-509. For an explanation of the benefits of these object modes, see "Benefits of Object Modes" on page 1-521.

Error Options for Filtering Mode

When you change the object mode of a constraint or unique index to the filtering mode, you can specify the following error options: WITHOUT ERROR or WITH ERROR.

The WITHOUT ERROR Option

The WITHOUT ERROR option signifies that when the database server executes an INSERT, DELETE, or UPDATE statement, and one or more of the target rows causes a constraint violation or unique-index violation, no integrity-violation error message is returned to the user. The WITHOUT ERROR option is the default error option.

The WITH ERROR Option

The WITH ERROR option signifies that when the database server executes an INSERT, DELETE, or UPDATE statement, and one or more of the target rows causes a constraint violation or unique-index violation, an integrity-violation error message is returned to the user.

Scope of Error Options

The WITH ERROR and WITHOUT ERROR options apply only when the database server executes an INSERT, DELETE, or UPDATE statement, and one or more of the target rows causes a constraint violation or unique index violation. These error options control whether the database server displays an integrity-violation error message after it executes these statements.

These error options do not apply when you attempt to change the object mode of a disabled constraint or disabled unique index to the enabled or filtering mode, and the SET statement fails because one or more rows in the target table violates the constraint or the unique-index requirement. In these cases, if a violations table has been started for the table that contains the inconsistent data, the database server returns an integrity-violation error message regardless of the error option that is specified in the SET statement.

Violations and Diagnostics Tables for Filtering Mode

When you specify the filtering mode for constraints or unique indexes in a SET statement, violations and diagnostics tables are not automatically started for the target table. When you set objects to the filtering mode, be sure to start the violations and diagnostics tables for the target table on which the filtering mode objects are defined. The violations table captures rows that fail to meet integrity requirements. The diagnostics table captures information about each row that fails to meet integrity requirements.

When to Start the Violations and Diagnostics Tables

You are not required to start the violations and diagnostics tables before you set objects to the filtering mode. If you have not started a violations and diagnostics table when you set an object to the filtering mode, the database server executes your SET statement and does not return an error. Similarly, if you issue an INSERT, DELETE, or UPDATE statement on the target table, and you have not started a violations and diagnostics table for the target table, the database server executes the statement and does not return an error as long as all of the integrity requirements on the table are satisfied.

If you have not started a violations and diagnostics table for the target table with filtering-mode objects, the database server does not return an error until an INSERT, DELETE, or UPDATE statement fails to satisfy an integrity requirement on the table. If an INSERT, DELETE, or UPDATE statement fails to satisfy the constraint or unique-index requirement for a particular row, the database server cannot filter the bad row to the violations table because no violations table is associated with the target table. The user receives an error message indicating that no violations table has been started for the target table.

To prevent such errors, start the violations and diagnostics tables for the target table at one of the following points:

- You can start the violations and diagnostics tables before you set any objects that are defined on the table to the filtering mode.
- You can start the violations and diagnostics tables after you set objects to the filtering mode but before any users issue INSERT, DELETE, or UPDATE statements that could violate any integrity requirements on the target table.

How to Start the Violations and Diagnostics Tables

To create the violations and diagnostics tables and associate them with the target table, use the START VIOLATIONS TABLE statement. In this statement, specify the name of the target table for which the violations and diagnostics tables are to be started. You can also assign names to the violations and diagnostics tables in this statement.

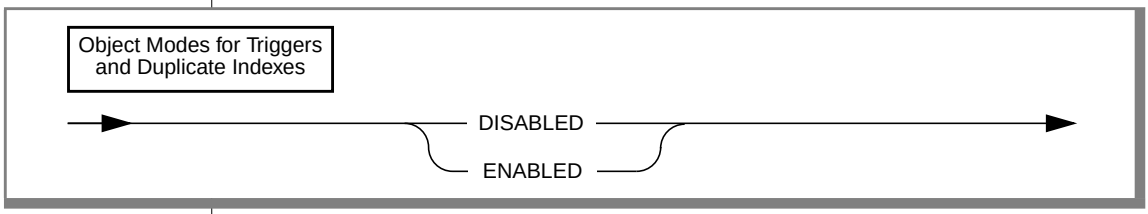
For further information on the START VIOLATIONS TABLE statement and the structure of the violations and diagnostics tables themselves, see the START VIOLATIONS TABLE statement on page 1-584.

How to Stop the Violations and Diagnostics Tables

After you turn off filtering mode for the objects that are defined on a target table, and you no longer need the violations and diagnostics tables, use the STOP VIOLATIONS TABLE statement to drop the association between the target table and the violations and diagnostics tables. In this statement, you specify the name of the target table whose association with the violations and diagnostics tables is to be dropped.

For further information on using the STOP VIOLATIONS TABLE statement, see the STOP VIOLATIONS TABLE statement on page 1-603.

Object Modes for Triggers and Duplicate Indexes



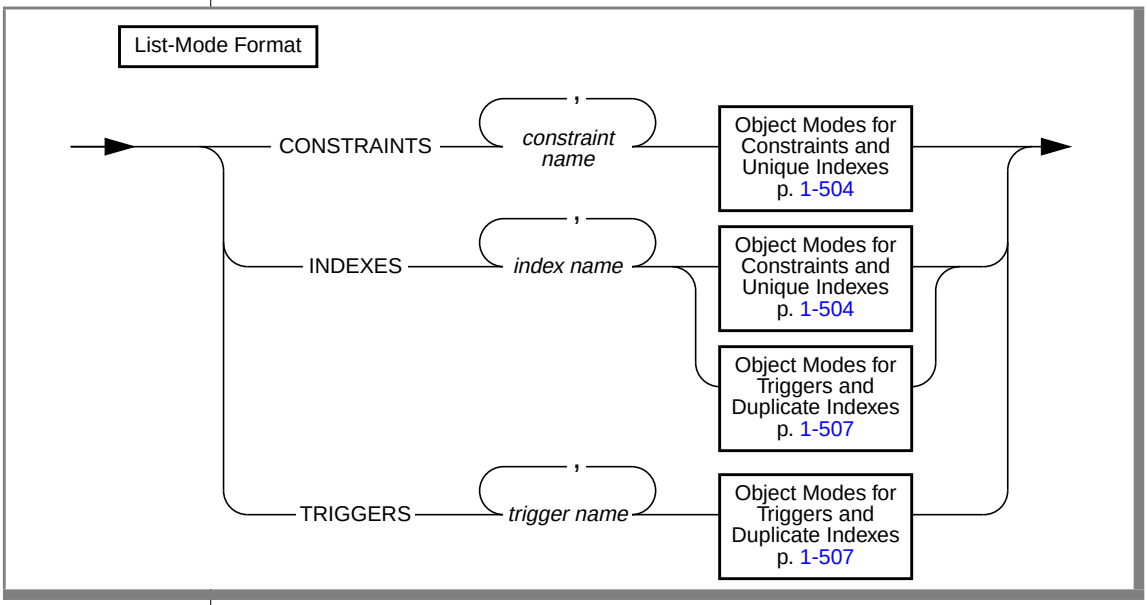
You can specify the disabled or enabled object modes for triggers or duplicate indexes. You must specify one of these object modes in your SET statement. The SET statement has no default object mode.

You can also specify the object mode for a trigger when you create the trigger with the CREATE TRIGGER statement. If you do not specify the object mode for a trigger in the CREATE TRIGGER statement or in a SET statement, the trigger is in the enabled object mode by default.

You can also specify the object mode for a duplicate index when you create the index with the CREATE INDEX statement. If you do not specify the object mode for a duplicate index in the CREATE INDEX statement or in a SET statement, the duplicate index is in the enabled object mode by default.

For definitions of the disabled and enabled object modes, see “Using Object Modes with Data Manipulation Statements” on page 1-509. For an explanation of the benefits of these two object modes, see “Benefits of Object Modes” on page 1-521.

List-Mode Format



Element	Purpose	Restrictions	Syntax
constraint name	The name of the constraint whose object mode is to be set, or a list of constraint names. There is no default value.	Each constraint in the list must be a local constraint. All constraints in the list must be defined on the same table.	Identifier, p. 1-723
index name	The name of the index whose object mode is to be set, or a list of index names. There is no default value.	Each index in the list must be a local index. All indexes in the list must be defined on the same table.	Identifier, p. 1-723
trigger name	The name of the trigger whose object mode is to be set, or a list of trigger names. There is no default value.	Each trigger in the list must be a local trigger. All triggers in the list must be defined on the same table.	Identifier, p. 1-723

Use the list-mode format to change the object mode for a particular constraint, index, or trigger. For example, to change the object mode of the unique index **unq_ssn** on the **cust_subset** table to filtering mode, enter the following statement:

```
SET INDEXES unq_ssn FILTERING
```

You can also use the list-mode format to change the object mode for a list of constraints, indexes, or triggers that are defined on the same table. Assume that four triggers are defined on the **cust_subset** table: **insert_trig**, **update_trig**, **delete_trig**, and **execute_trig**. Also assume that all four triggers are in the enabled mode. To change the object mode of all the triggers except **execute_trig** to the disabled mode, enter the following statement:

```
SET TRIGGERS insert_trig, update_trig, delete_trig DISABLED
```

Using Object Modes with Data Manipulation Statements

You can use object modes to control the effects of INSERT, DELETE, and UPDATE statements. Your choice of object modes affects the tables whose data you are manipulating, the behavior of the objects defined on those tables, and the behavior of the data manipulation statements themselves.

What do we mean by the terms *enabled*, *disabled*, and *filtering*? Definitions of these object modes follow. These definitions explain how each object mode affects tables and data manipulation statements. The definitions focus on the object modes of constraints as an illustration, but the same principles apply to indexes and triggers as well.

Definition of Enabled Mode

Constraints, indexes, and triggers are in the enabled mode by default. When an object is in the enabled mode, the database server recognizes the existence of the object and takes the object into consideration while it executes data manipulation statements. For example, when a constraint is enabled, any INSERT, UPDATE, or DELETE statement that violates the constraint fails, and the target row remains unchanged. In addition, the user receives an error message.

Definition of Disabled Mode

When an object is in the disabled mode, the database server acts as if the object did not exist and does not take it into consideration during the execution of data manipulation statements. For example, when a constraint is disabled, any INSERT, UPDATE, or DELETE statement that violates the constraint succeeds, and the target row is changed. The user does not receive an error message.

Definition of Filtering Mode

When an object is in the filtering mode, the object behaves the same as in the enabled mode in that the database server recognizes the existence of the object during INSERT, UPDATE, and DELETE statements. For example, when a constraint is in the filtering mode, and an INSERT, DELETE, or UPDATE statement is executed, any target rows that violate the constraint remain unchanged.

However, the database server handles data manipulation statements differently for objects in enabled and filtering mode, as the following paragraphs describe:

- If a constraint or unique index is in the enabled mode, the database server carries out the INSERT, UPDATE, or DELETE statement only if all the target rows affected by the statement satisfy the constraint or the unique index requirement. The database server updates all the target rows in the table.
- If a constraint or unique index is in the filtering mode, the database server carries out the INSERT, UPDATE, or DELETE statement even if one or more of the target rows fail to satisfy the constraint or the unique index requirement. The database server updates the good rows in the table (the target rows that satisfy the constraint or unique index requirement). The database server does not update the bad rows in the table (that is, the target rows that fail to satisfy the constraint or unique index requirement). Instead the database server sends each bad row to a special table called the violations table. The database server places information about the nature of the violation for each bad row in another special table called the diagnostics table.

Example of Object Modes with Data Manipulation Statements

An example with the INSERT statement can illustrate the differences between the enabled, disabled, and filtering modes. Consider an INSERT statement in which a user tries to add a row that does not satisfy an integrity constraint on a table. For example, assume that a user **joe** has created a table named **cust_subset**, and this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives). The **ssn** column has the INT data type. The other three columns have the CHAR data type.

Assume that user **joe** has defined the **lname** column as not null but has not assigned a name to the not null constraint, so the database server has implicitly assigned the name **n104_7** to this constraint. Finally, assume that user **joe** has created a unique index named **unq_ssn** on the **ssn** column.

Now a user **linda** who has the Insert privilege on the **cust_subset** table enters the following INSERT statement on this table:

```
INSERT INTO cust_subset (ssn, fname, city)
    VALUES (973824499, "jane", "los altos")
```

User **linda** has entered values for all the columns of the new row except for the **lname** column, even though the **lname** column has been defined as a not null column. The database server behaves in the following ways, depending on the object mode of the constraint:

- If the constraint is disabled, the row is inserted in the target table, and no error is returned to the user.
- If the constraint is enabled, the row is not inserted in the target table. A constraint-violation error is returned to the user, and the effects of the statement are rolled back (if the database is an OnLine database with logging).
- If the constraint is filtering, the row is not inserted in the target table. Instead the row is inserted in the violations table. Information about the integrity violation caused by the row is placed in the diagnostics table. The effects of the INSERT statement are not rolled back. You receive an error message if you specified the WITH ERROR option for the filtering-mode constraint. By analyzing the contents of the violations and the diagnostics tables, you can identify the reason for the failure and either take corrective action or roll back the operation.

We can better grasp the distinctions among disabled, enabled, and filtering modes by viewing the actual results of the INSERT statement shown in the preceding example.

Results of the Insert Operation When the Constraint Is Disabled

If the not null constraint on the **cust_subset** table is disabled, the INSERT statement that user **linda** issues successfully inserts the new row in this table. The new row of the **cust_subset** table has the following column values.

ssn	fname	lname	city
973824499	jane	NULL	los altos

Results of the Insert Operation When the Constraint Is Enabled

If the not null constraint on the **cust_subset** table is enabled, the INSERT statement fails to insert the new row in this table. Instead user **linda** receives the following error message when she enters the INSERT statement:

```
-292 An implied insert column lname does not accept NULLs.
```

Results of the Insert When Constraint Is in Filtering Mode

If the not null constraint on the **cust_subset** table is set to the filtering mode, the INSERT statement that user **linda** issues fails to insert the new row in this table. Instead the new row is inserted into the violations table, and a diagnostic row that describes the integrity violation is added to the diagnostics table.

Assume that user **joe** has started a violations and diagnostics table for the **cust_subset** table. The violations table is named **cust_subset_vio**, and the diagnostics table is named **cust_subset_dia**. The new row added to the **cust_subset_vio** violations table when user **linda** issues the INSERT statement on the **cust_subset** target table has the following column values.

ssn	fname	lname	city	informix_tupleid	informix_optype	informix_recowner
973824499	jane	NULL	los altos	1	I	linda

This new row in the **cust_subset_vio** violations table has the following characteristics:

- The first four columns of the violations table exactly match the columns of the target table. These four columns have the same names and the same data types as the corresponding columns of the target table, and they have the column values that were supplied by the INSERT statement that user **linda** entered.
- The value `1` in the **informix_tupleid** column is a unique serial identifier that is assigned to the nonconforming row.
- The value `I` in the **informix_optype** column is a code that identifies the type of operation that has caused this nonconforming row to be created. Specifically, `I` stands for an insert operation.
- The value `linda` in the **informix_recowner** column identifies the user who issued the statement that caused this nonconforming row to be created.

The INSERT statement that user **linda** issued on the **cust_subset** target table also causes a diagnostic row to be added to the **cust_subset_dia** diagnostics table. The new diagnostic row added to the diagnostics table has the following column values.

informix_tupleid	objtype	objowner	objname
1	C	joe	n104_7

This new diagnostic row in the **cust_subset_dia** diagnostics table has the following characteristics:

- This row of the diagnostics table is linked to the corresponding row of the violations table by means of the **informix_tupleid** column that appears in both tables. The value `1` appears in this column in both tables.
- The value `C` in the **objtype** column identifies the type of integrity violation that the corresponding row in the violations table caused. Specifically, the value `C` stands for a constraint violation.

- The value joe in the **objowner** column identifies the owner of the constraint for which an integrity violation was detected.
- The value n104_7 in the **objname** column gives the name of the constraint for which an integrity violation was detected.

By joining the violations and diagnostics tables, user **joe** (who owns the **cust_subset** target table and its associated special tables) or the DBA can find out that the row in the violations table whose **informix_tupleid** value is 1 was created after an INSERT statement and that this row is violating a constraint. The table owner or DBA can query the **sysconstraints** system catalog table to determine that this constraint is a not null constraint. Now that the reason for the failure of the INSERT statement is known, user **joe** or the DBA can take corrective action.

Multiple Diagnostic Rows for One Violations Row

In the preceding example, only one row in the diagnostics table corresponds to the new row in the violations table. However, more than one diagnostic row can be added to the diagnostics table when a single new row is added to the violations table. For example, if the **ssn** value (973824499) that user **linda** entered in the INSERT statement had been the same as an existing value in the **ssn** column of the **cust_subset** target table, only one new row would appear in the violations table, but the following two diagnostic rows would be present in the **cust_subset_dia** diagnostics table.

informix_tupleid	objtype	objowner	objname
1	C	joe	n104_7
1	I	joe	unq_ssn

Both rows in the diagnostics table correspond to the same row of the violations table because both of these rows have the value 1 in the **informix_tupleid** column. However, the first diagnostic row identifies the constraint violation caused by the INSERT statement that user **linda** issued, while the second diagnostic row identifies the unique-index violation caused by the same INSERT statement. In this second diagnostic row, the value I in the **objtype** column stands for a unique-index violation, and the value unq_ssn in the **objname** column gives the name of the index for which the integrity violation was detected.

For information on when and how to start violations and diagnostics tables for a target table, see "Violations and Diagnostics Tables for Filtering Mode" on page 1-505. For further information on the structure of the violations and diagnostics tables, see the START VIOLATIONS TABLE statement on page 1-584.

Using Object Modes to Achieve Data Integrity

In addition to using object modes with data manipulation statements, you can also use object modes when you add a new constraint or new unique index to a target table. By selecting the correct object mode, you can add the constraint or index to the target table easily even if existing rows in the target table violate the new integrity specification.

You can add a new constraint or index easily by taking the following steps. If you follow this procedure, you do not have to examine the entire target table to identify rows that fail to satisfy the constraint or unique-index requirement:

- Add the constraint or index in the enabled mode. If all existing rows in the table satisfy the constraint or unique-index requirement, your ALTER TABLE or CREATE INDEX statement executes successfully, and you do not need to take any further steps. However, if any existing rows in the table fail to satisfy the constraint or unique-index requirement, your ALTER TABLE or CREATE INDEX statement returns an error message, and you need to take the following steps.
- Add the constraint or index in the disabled mode. Issue the ALTER TABLE statement again, and specify the DISABLED keyword in the ADD CONSTRAINT or MODIFY clause; or issue the CREATE INDEX statement again, and specify the DISABLED keyword.
- Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.
- Issue a SET statement to switch the object mode of the constraint or index to the enabled mode. When you issue this statement, the statement fails, and existing rows in the target table that violate the constraint or the unique-index requirement are duplicated in the violations table. The constraint or index remains disabled, and you receive an integrity-violation error message.

- Issue a SELECT statement on the violations table to retrieve the nonconforming rows that were duplicated from the target table. You might need to join the violations and diagnostics tables to get all the necessary information.
- Take corrective action on the rows in the target table that violate the constraint.
- After you fix all the nonconforming rows in the target table, issue the SET statement again to switch the disabled constraint or index to the enabled mode. This time the constraint or index is enabled, and no integrity-violation error message is returned because all rows in the target table now satisfy the new constraint or unique-index requirement.

Example of Using Object Modes to Achieve Data Integrity

The following example shows how to use object modes to add a constraint and unique index to a target table easily. Assume that a user **joe** has created a table named **cust_subset**, and this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives).

Also assume that no constraints or unique indexes are defined on the **cust_subset** table and that the **fname** column is the primary key. In addition, assume that no violations and diagnostics tables currently exist for this target table. Finally, assume that this table currently contains four rows with the following column values.

ssn	fname	lname	city
111763227	mark	jackson	sunnyvale
222781244	rhonda	NULL	palo alto
111763227	steve	NULL	san mateo
333992276	tammy	jones	san jose

Adding the Objects in the Enabled Mode

User **joe**, the owner of the **cust_subset** table, enters the following statements to add a unique index on the **ssn** column and a not null constraint on the **lname** column:

```
CREATE UNIQUE INDEX unq_ssn ON cust_subset (ssn) ENABLED;
ALTER TABLE cust_subset MODIFY (lname CHAR(15)
    NOT NULL CONSTRAINT lname_notblank ENABLED);
```

Both of these statements fail because existing rows in the **cust_subset** table violate the integrity specifications. The row whose **fname** value is `rhonda` violates the not null constraint on the **lname** column. The row whose **fname** value is `steve` violates both the not null constraint on the **lname** column and the unique-index requirement on the **ssn** column.

Adding the Objects in the Disabled Mode

To recover from the preceding errors, user **joe** reenters the CREATE INDEX and ALTER TABLE statements and specifies the disabled mode in both statements, as follows:

```
CREATE UNIQUE INDEX unq_ssn ON cust_subset (ssn) DISABLED;
ALTER TABLE cust_subset MODIFY (lname CHAR(15)
    NOT NULL CONSTRAINT lname_notblank DISABLED);
```

Both of these statements execute successfully because the database server does not enforce unique-index requirements or constraint specifications when these objects are disabled.

Starting a Violations and Diagnostics Table

Now that the new constraint and index are added for the **cust_subset** table, user **joe** takes steps to find out which existing rows in the **cust_subset** table violate the constraint and the index.

First, user **joe** enters the following statement to start a violations and diagnostics table for the **cust_subset** table:

```
START VIOLATIONS TABLE FOR cust_subset
```

Because user **joe** has not assigned names to the violations and diagnostics tables in this statement, the tables are named **cust_subset_vio** and **cust_subset_dia** by default.

Using the SET Statement to Capture Violations

Now that violations and diagnostics tables exist for the target table, user **joe** issues the following SET statement to switch the mode of the new index and constraint from the disabled mode to the enabled mode:

```
SET CONSTRAINTS, INDEXES FOR cust_subset ENABLED
```

The result of this SET statement is that the existing rows in the **cust_subset** table that violate the constraint and the unique-index requirement are copied to the **cust_subset_vio** violations table, and diagnostic information about the nonconforming rows is added to the **cust_subset_dia** diagnostics table. The SET statement fails, and the constraint and index remain disabled.

The following table shows the contents of the **cust_subset_vio** violations table after user **joe** issues the SET statement.

ssn	fname	lname	city	informix_tupleid	informix_optype	informix_recowner
222781244	rhonda	NULL	palo alto	1	S	joe
111763227	steve	NULL	san mateo	2	S	joe

These two rows in the **cust_subset_vio** violations table have the following characteristics:

- The row in the **cust_subset** target table whose **fname** value is `rhonda` is duplicated to the **cust_subset_vio** violations table because this row violates the not null constraint on the **lname** column.
- The row in the **cust_subset** target table whose **fname** value is `steve` is duplicated to the **cust_subset_vio** violations table because this row violates the not null constraint on the **lname** column and the unique-index requirement on the **ssn** column.
- The value `1` in the **informix_tupleid** column for the first row and the value `2` in the **informix_tupleid** column for the second row are unique serial identifiers assigned to the nonconforming rows.

- The value S in the **informix_optype** column for each row is a code that identifies the type of operation that has caused this nonconforming row to be placed in the violations table. Specifically, the S stands for a SET statement.
- The value joe in the **informix_recowner** column for each row identifies the user who issued the statement that caused this nonconforming row to be placed in the violations table.

The following table shows contents of the **cust_subset_dia** diagnostics table after user **joe** issues the SET statement.

informix_tupleid	objtype	objowner	objname
1	C	joe	lname_notblank
2	C	joe	lname_notblank
2	I	joe	unq_ssn

These three rows in the **cust_subset_dia** diagnostics table have the following characteristics:

- Each row in the diagnostics table and the corresponding row in the violations table are joined by the **informix_tupleid** column that appears in both tables.
- The first row in the diagnostics table has an **informix_tupleid** value of 1. It is joined to the row in the violations table whose **informix_tupleid** value is 1. The value C in the **objtype** column for this diagnostic row identifies the type of integrity violation that was caused by the corresponding row in the violations table. Specifically, the value C stands for a constraint violation. The value lname_notblank in the **objname** column for this diagnostic row gives the name of the constraint for which an integrity violation was detected.

- The second row in the diagnostics table has an **informix_tupleid** value of 2. It is joined to the row in the violations table whose **informix_tupleid** value is 2. The value C in the **objtype** column for this second diagnostic row indicates that a constraint violation was caused by the corresponding row in the violations table. The value lname_notblank in the **objname** column for this diagnostic row gives the name of the constraint for which an integrity violation was detected.
- The third row in the diagnostics table has an **informix_tupleid** value of 2. It is also joined to the row in the violations table whose **informix_tupleid** value is 2. The value I in the **objtype** column for this third diagnostic row indicates that a unique-index violation was caused by the corresponding row in the violations table. The value unq_ssn in the **objname** column for this diagnostic row gives the name of the index for which an integrity violation was detected.
- The value joe in the **objowner** column of all three diagnostic rows identifies the owner of the object for which an integrity violation was detected. The name of user **joe** appears in all three rows because he created the constraint and index on the **cust_subset** table.

Identifying Nonconforming Rows to Obtain Information

To determine the contents of the violations table, user **joe** enters a SELECT statement to retrieve all rows from the table. Then, to obtain complete diagnostic information about the nonconforming rows, user **joe** joins the violations and diagnostics tables by means of another SELECT statement. User **joe** can perform these operations either interactively or through a program.

Taking Corrective Action on the Nonconforming Rows

After the user **joe** identifies the nonconforming rows in the **cust_subset** table, he can correct them. For example, he can enter UPDATE statements on the **cust_subset** table either interactively or through a program.

Enabling the Disabled Objects

Once all the nonconforming rows in the **cust_subset** table are corrected, user **joe** issues the following SET statement to set the new constraint and index to the enabled mode:

```
SET CONSTRAINTS, INDEXES FOR cust_subset ENABLED
```

This time the SET statement executes successfully. The new constraint and new unique index are enabled, and no error message is returned to user **joe** because all rows in the **cust_subset** table now satisfy the new constraint specification and unique-index requirement.

Benefits of Object Modes

The preceding examples show how object modes work when users execute data manipulation statements on target tables or add new constraints and indexes to target tables. The preceding examples suggest some of the benefits of the different object modes. The following sections state these benefits explicitly.

Benefits of Disabled Mode

The benefits of the disabled mode are as follows:

- You can use the disabled mode to insert many rows quickly into a target table. Especially during load operations, updates of the existing indexes and enforcement of referential constraints make up a big part of the total cost of the operation. By disabling the indexes and referential constraints during the load operation, you improve the performance and efficiency of the load.

- To add a new constraint or new unique index to an existing table, you can add the object even if some rows in the table do not satisfy the new integrity specification. If the constraint or index is added to the table in disabled mode, your ALTER TABLE or CREATE INDEX statement does not fail no matter how many existing rows violate the new integrity requirement.

  If a violations table has been started, a SET statement that switches the disabled objects to the enabled or filtering mode fails, but it causes the nonconforming rows in the target table to be duplicated in the violations table so that you can identify the rows and take corrective action. After you fix the nonconforming rows in the target table, you can reissue the SET statement to switch the disabled objects to the enabled or filtering mode.

Benefits of Enabled Mode

The enabled mode is the default object mode for all database objects. We can summarize the benefits of this mode for each type of database object as follows:

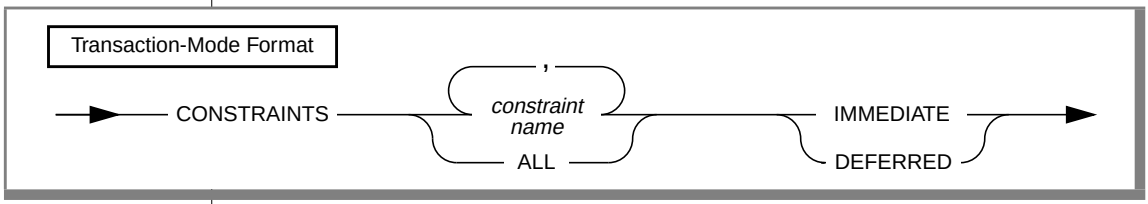
- The benefit of enabled mode for constraints is that the database server enforces the constraint and thus ensures the consistency of the data in the database.
- The benefit of enabled mode for indexes is that the database server updates the index after insert, delete, and update operations. Thus the index is up to date and is used by the optimizer during database queries.
- The benefit of enabled mode for triggers is that the trigger event always sets the triggered action in motion. Thus the purpose of the trigger is always realized during actual data-manipulation operations.

Benefits of Filtering Mode

The benefits of setting a constraint or unique index to the filtering mode are as follows:

- During load operations, inserts that violate a filtering mode constraint or unique index do not cause the load operation to fail. Instead, the database server filters the bad rows to the violations table and continues the load operation.
- When an INSERT, DELETE, or UPDATE statement that affects multiple rows causes a filtering mode constraint or unique index to be violated for a particular row or rows, the statement does not fail. Instead, the database server filters the bad row or rows to the violations table and continues to execute the statement.
- When any INSERT, DELETE, or UPDATE statement violates a filtering mode constraint or unique index, the user can identify the failed row or rows and take corrective action. The violations and diagnostics tables capture the necessary information, and users can take corrective action after they analyze this information.

Transaction-Mode Format



Element	Purpose	Restrictions	Syntax
constraint name	The name of the constraint whose transaction mode is to be changed, or a list of constraint names. There is no default value.	The specified constraint must exist in an OnLine database with logging. You cannot change the transaction mode of a constraint to deferred mode unless the constraint is currently in the enabled mode. All constraints in a list of constraints must exist in the same database.	Identifier, p. 1-723

You can use the transaction-mode format of the SET statement to set the transaction mode of constraints.

You use the IMMEDIATE keyword to set the transaction mode of constraints to statement-level checking. You use the DEFERRED keyword to set the transaction mode to transaction-level checking.

You can set the transaction mode of constraints only in an OnLine database with logging.

Statement-Level Checking

When you set the transaction mode to immediate, statement-level checking is turned on, and all specified constraints are checked at the end of each INSERT, UPDATE, or DELETE statement. If a constraint violation occurs, the statement is not executed. Immediate is the default transaction mode of constraints.

Transaction-Level Checking

When you set the transaction mode of constraints to deferred, statement-level checking is turned off, and all specified constraints are not checked until the transaction is committed. If a constraint violation occurs while the transaction is being committed, the transaction is rolled back.

Tip: *If you defer checking a primary-key constraint, the checking of the not null constraint for that column or set of columns is also deferred.*

Duration of Transaction Modes

The duration of the transaction mode that the SET statement specifies is the transaction in which the SET statement is executed. You cannot execute this form of the SET statement outside a transaction. Once a COMMIT WORK or ROLLBACK WORK statement is successfully completed, the transaction mode of all constraints reverts to IMMEDIATE.

Switching Transaction Modes

To switch from transaction-level checking to statement-level checking, you can use the SET statement to set the transaction mode to immediate, or you can use a COMMIT WORK or ROLLBACK WORK statement in your transaction.

Specifying All Constraints or a List of Constraints

You can specify all constraints in the database in your SET statement, or you can specify a single constraint or list of constraints.

Specifying All Constraints

If you specify the ALL keyword, the SET statement sets the transaction mode for all constraints in the database. If any statement in the transaction requires that any constraint on any table in the database be checked, the database server performs the checks at the statement level or the transaction level, depending on the setting that you specify in the SET statement.

Specifying a List of Constraints

If you specify a single constraint name or a list of constraints, the SET statement sets the transaction mode for the specified constraints only. If any statement in the transaction requires checking of a constraint that you did not specify in the SET statement, that constraint is checked at the statement level regardless of the setting that you specified in the SET statement for other constraints.

When you specify a list of constraints, the constraints do not have to be defined on the same table, but they must exist in the same database.

Specifying Remote Constraints

You can set the transaction mode of local constraints or remote constraints. That is, the constraints that are specified in the transaction-mode form of the SET statement can be constraints that are defined on local tables or constraints that are defined on remote tables.

Examples of Setting the Transaction Mode for Constraints

The following example shows how to defer checking constraints within a transaction until the transaction is complete. The SET CONSTRAINTS statement in the example specifies that any constraints on any tables in the database are not checked until the COMMIT WORK statement is encountered.

```
BEGIN WORK
SET CONSTRAINTS ALL DEFERRED
.
.
.
COMMIT WORK
```

The following example specifies that a list of constraints is not checked until the transaction is complete:

```
BEGIN WORK
SET CONSTRAINTS update_const, insert_const DEFERRED
.
.
.
COMMIT WORK
```

References

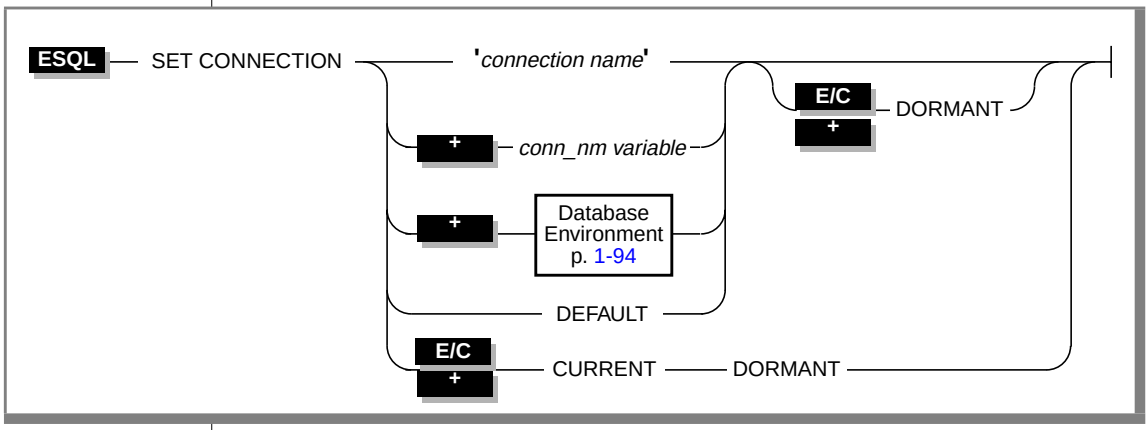
See the START VIOLATIONS TABLE and STOP VIOLATIONS TABLE statements in this manual.

For information on the system catalog tables associated with the SET statement, see the **sysobjstate** and **sysviolations** tables in the *Informix Guide to SQL: Reference*.

SET CONNECTION

The SET CONNECTION statement reestablishes a connection between an application and a database environment and makes the connection current. You can also use the SET CONNECTION statement with the DORMANT option to put the current connection in a dormant state.

Syntax



Element	Purpose	Restrictions	Syntax
connection name	Quoted string that identifies the *connection name* that you assigned to a specific connection. It is the *connection name* assigned by the CONNECT statement when the initial connection was made.	The database must already exist. If you use the SET CONNECTION statement with the DORMANT option, *connection name* must represent the current connection. If you use the SET CONNECTION statement without the DORMANT option, *connection name* must represent a dormant connection.	Quoted String, p. 1-757
conn_nm variable	Host variable that contains the value of *connection name*	Variable must be the character data type.	Variable name must conform to language-specific rules for variable names.

Usage

You can use the SET CONNECTION statement to change the state of a connection in the following ways:

- Make a dormant connection current

  For information on using SET CONNECTION to make a dormant connection current, see "Making a Dormant Connection the Current Connection" below.

- Make the current connection dormant

  For information on using SET CONNECTION to make the current connection dormant, see "Making a Current Connection Dormant" on page 1-529.

Making a Dormant Connection the Current Connection

The SET CONNECTION statement, with no DORMANT option, makes the specified dormant connection the current one. The connection that the application specifies must be dormant. The connection that is current when the statement executes becomes dormant. A dormant connection is a connection that has been established but is not current.

The SET CONNECTION statement in the following example makes connection `con1` the current connection and makes `con2` a dormant connection:

```
CONNECT TO 'stores7' AS 'con1'
...
CONNECT TO 'demo7' AS 'con2'
...
SET CONNECTION 'con1'
```

A dormant connection has a *connection context* associated with it. When an application makes a dormant connection current, it reestablishes that connection to a database environment and restores its connection context. (For more information on connection context, see page 1-91.) Reestablishing a connection is comparable to establishing the initial connection, except that it typically avoids authenticating the user's permissions again, and it saves reallocating resources associated with the initial connection. For example, the application does not need to reprepare any statements that have previously been prepared in the connection nor does it need to redeclare any cursors.

Making a Current Connection Dormant

The SET CONNECTION statement with the DORMANT option makes the specified current connection a dormant connection. For example, the following SET CONNECTION statement makes connection `con1` dormant:

```
SET CONNECTION 'con1' DORMANT
```

The SET CONNECTION statement with the DORMANT option generates an error if you specify a connection that is already dormant. For example, if connection `con1` is current and connection `con2` is dormant, the following SET CONNECTION statement returns an error message:

```
SET CONNECTION 'con2' DORMANT
```

However, the following SET CONNECTION statement executes successfully:

```
SET CONNECTION 'con1' DORMANT
```

Dormant Connections in a Single-Threaded Environment

In a single-threaded application (an ESQL/C application that does not use threads or an ESQL/COBOL application), the DORMANT option makes the current connection dormant. The availability of the DORMANT option in single-threaded applications makes single-threaded ESQL/C applications upwardly compatible with thread-safe ESQL/C applications.

Dormant Connections in a Thread-Safe ESQL/C Environment

E/C

As in a single-threaded application, a thread-safe ESQL/C application (an ESQL/C application that uses threads) can establish many connections to one or more databases. However, in the single-threaded environment, only one connection can be active while the program executes. In the thread-safe environment, there can be many threads (concurrent pieces of work performing particular tasks) in one ESQL/C application, and each thread can have one active connection.

An active connection is associated with a particular thread. Two threads cannot share the same active connection. Once a thread makes an active connection dormant, that connection is available to other threads. A dormant connection is still established but is not currently associated with any thread. For example, if the connection named `con1` is active in the thread named `thread_1`, the thread named `thread_2` cannot make connection `con1` its active connection until `thread_1` has made connection `con1` dormant.

In a thread-safe ESQL/C application, the DORMANT option makes an active connection dormant. Another thread can now use the connection by issuing the SET CONNECTION statement without the DORMANT option.

The following code fragment from a thread-safe ESQL/C program shows how a particular thread within a thread-safe application makes a connection active, performs work on a table through this connection, and then makes the connection dormant so that other threads can use the connection:

```
thread_2()
{    /* Make con2 an active connection */
     EXEC SQL connect to 'db2' as 'con2';
     /*Do insert on table t2 in db2*/
     EXEC SQL insert into table t2 values(10);
     /* make con2 available to other threads */
     EXEC SQL set connection 'con2' dormant;
}
.
.
.
```

If a connection to a database environment is initiated with the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, an ongoing transaction can used by any thread that subsequently connects to that database environment. In addition, if an open cursor is associated with such a connection, the cursor remains open when the connection is made dormant. Threads within a thread-safe ESQL/C application can use the same cursor by making the associated connection current even though only one thread can use the connection at any given time.

For a detailed discussion of thread-safe ESQL/C applications and the use of the SET CONNECTION statement in these applications, see Chapter 11 of the *INFORMIX-ESQL/C Programmer's Manual*. ♦

Identifying the Connection

If the application did not use *connection name* in the initial CONNECT statement, you must use a database environment (such as a database name or a database pathname) as the connection name. For example, the following SET CONNECTION statement uses a database environment for the connection name because the CONNECT statement does not use *connection name*. For information about quoted strings that contain a database environment , see "Database Environment" on page 1-94.

```
CONNECT TO 'stores7'
...
CONNECT TO 'demo7'
...
SET CONNECTION 'stores7'
```

If a connection to a database server was assigned a *connection name*, however, you must use the connection name to reconnect to the database server. An error is returned if you use a database environment rather than the connection name when a connection name exists.

The DEFAULT Option

Use the DEFAULT option to identify the default connection for a SET CONNECTION statement. The default connection is one of the following connections:

- An explicit default connection (a connection established with the CONNECT TO DEFAULT statement)
- An implicit default connection (any connection made using the DATABASE, CREATE DATABASE, or START DATABASE statements)

You can use SET CONNECTION without a DORMANT option to reestablish the default connection or with the DORMANT option to make the default connection dormant. See "The DEFAULT Option" on page 1-91 and "The Implicit Connection with DATABASE Statements" on page 1-92 for more information.

The CURRENT Keyword

Use the CURRENT keyword with the DORMANT option of the SET CONNECTION statement as a shorthand form of identifying the current connection. The CURRENT keyword replaces the current connection name. If the current connection is `con1`, the following two statements are equivalent:

```
SET CONNECTION 'con1' DORMANT;

SET CONNECTION CURRENT DORMANT;
```

When a Transaction is Active

When you issue a SET CONNECTION statement without the DORMANT option, the SET CONNECTION statement implicitly puts the current connection in the dormant state. When you issue a SET CONNECTION statement (with the DORMANT option), the SET CONNECTION statement explicitly puts the current connection in the dormant state. In either case, the statement can fail if a connection that becomes dormant has an uncommitted transaction.

If the connection that becomes dormant has an uncommitted transaction, the following conditions apply:

- If the connection was established with the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the SET CONNECTION statement succeeds and puts the connection in a dormant state.
- If the connection was established without the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the SET CONNECTION statement fails and cannot set the connection to a dormant state and the transaction in the current connection continues to be active. The statement generates an error and the application must decide whether to commit or roll back the active transaction.

When Current Connection Is to INFORMIX-OnLine Dynamic Server Prior to Version 6.0

If the current connection is to a version of the OnLine database server prior to 6.0, the following conditions apply when a SET CONNECTION statement with or without the DORMANT option executes:

- If the connection to be made dormant was established using the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the application *can* switch to a different connection.
- If the connection to be made dormant was established without the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, the application *cannot* switch to a different connection; the SET CONNECTION statement returns an error. The application must use the CLOSE DATABASE statement to close the database and drop the connection.

References

See the CONNECT, DISCONNECT, and DATABASE statements in this manual.

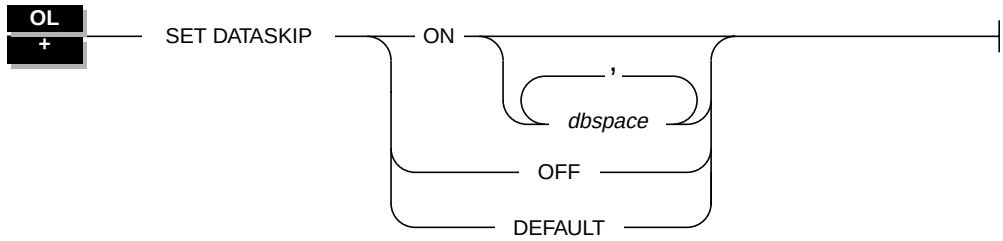
In the *INFORMIX-ESQL/C Programmer's Manual*, see the discussions of the SET CONNECTION statement and thread-safe applications.

SET DATASKIP

The SET DATASKIP statement allows you to control whether OnLine skips a dbspace that is unavailable (for example, due to a media failure) in the course of processing a transaction.

Syntax

OL +



Element	Purpose	Restrictions	Syntax
dbspace	The name of the skipped dbspace	The dbspace must exist at the time the statement is executed.	Identifier, p. 1-723

Usage

Use the SET DATASKIP statement to instruct the database server to skip a dbspace that is unavailable during the course of processing a transaction.

ESQL

You receive a warning if a dbspace is skipped. The warning flag **sqlca.sqlwarn.sqlwarn6** is set to `W` if a dbspace is skipped. For more information about this topic, see the *INFORMIX-ESQL/C Programmer's Manual* or the *INFORMIX-ESQL/COBOL Programmer's Manual*. ♦

When you SET DATASKIP ON without specifying a dbspace, you are telling the database server to skip any dbspaces in the fragmentation list that are unavailable. You can use the **onstat -d** or **-D** utility to determine if a dbspace is down.

When you SET DATASKIP ON *dbspace*, you are telling the database server to skip the specified dbspace if it is unavailable.

Use the SET DATASKIP OFF statement to turn off the dataskip feature.

When the setting is DEFAULT, the database server uses the setting for the dataskip feature from the ONCONFIG file. The OnLine administrator can change the setting of the dataskip feature at runtime. See the *INFORMIX-OnLine Dynamic Server Administrator's Guide* for more information.

Under What Circumstances Is a Dbspace Skipped?

The database server skips a dbspace when SET DATASKIP is set to ON and the dbspace is unavailable.

The database server cannot skip a dbspace under certain conditions. The following list outlines those conditions:

- Referential constraint checking

  When you want to delete a parent row, the child rows must also be available for deletion. The child rows must exist in an available fragment.

  When you want to insert a new child table, the parent table must be found in the available fragments.

- Updates

  When you perform an update that moves a record from one fragment to another, both fragments must be available.

- Inserts

  When you try to insert records in a expression-based fragmentation strategy and the dbspace is unavailable, an error is returned. When you try to insert records in a round-robin fragment-based strategy, and a dbspace is down, the database server inserts the rows into any available dbspace. When no dbspace is available, an error is returned.

- Indexing

  When you perform updates that affect the index, such as when you insert or delete records, or when you update an indexed field, the index must be available.

  When you try to create an index, the dbspace you want to use must be available.

- Serial keys

  The first fragment is used to store the current serial-key value internally. This is not visible to you except when the first fragment becomes unavailable and a new serial key value is required, which happens during insert statements.

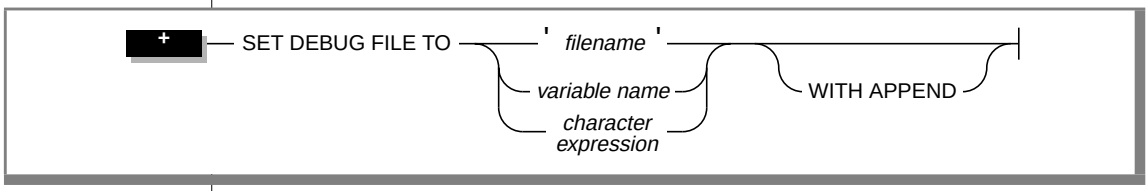
References

For additional information about how to set the dataskip feature in the ONCONFIG file and how to use the **onspaces** utility, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

SET DEBUG FILE TO

Use the SET DEBUG FILE TO statement to name the file that is to hold the run-time trace output of a stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
character expression	An expression that evaluates to a filename	The filename that is derived from the expression must be usable. The same restrictions apply to the derived filename as to the *filename* parameter.	Expression, p. 1-671
filename	A quoted string that identifies the pathname and filename of the file that contains the output of the TRACE statement. See "Location of the Output File" on page 1-539 for information on the default actions that are taken if you omit the pathname.	You can specify a new or existing file. If you specify an existing file, you must include the WITH APPEND keywords if you want to preserve the current contents of the file intact. See "Using the WITH APPEND Option" on page 1-538 for further information.	Quoted String, p. 1-757. The pathname and filename must conform to the conventions of your operating system.
variable name	A host variable that holds the value of *filename*	The host variable must be a character data type.	The name of the host variable must conform to language-specific rules for variable names.

Usage

This statement indicates that the output of the TRACE statement in the stored procedure goes to the file that *filename* indicates. Each time the TRACE statement is executed, the trace data is added to this output file.

Using the WITH APPEND Option

The output file that you specify in the SET DEBUG TO file statement can be a new file or existing file.

If you specify an existing file, the current contents of the file are purged when you issue the SET DEBUG TO FILE statement. The first execution of a TRACE command sends trace output to the beginning of the file.

However, if you include the WITH APPEND option, the current contents of the file are preserved when you issue the SET DEBUG TO FILE statement. The first execution of a TRACE command adds trace output to the end of the file.

If you specify a new file in the SET DEBUG TO FILE statement, it makes no difference whether you include the WITH APPEND option. The first execution of a TRACE command sends trace output to the beginning of the new file whether you include or omit the WITH APPEND option.

Closing the Output File

To close the file that the SET DEBUG FILE TO statement opened, issue another SET DEBUG FILE TO statement with another filename. You can then edit the contents of the first file.

Redirecting Trace Output

You can use the SET DEBUG FILE TO statement outside a procedure to direct the trace output of the procedure to a file. You also can use this statement inside a procedure to redirect its own output.

Location of the Output File

If you invoke a SET DEBUG FILE TO statement with a simple filename on a local database, the output file is located in your current directory. If your current database is on a remote database server, the output file is located in your home directory on the remote database server. If you provide a full pathname for the debug file, the file is placed in the directory and file that you specify on the remote database server. If you do not have write permissions in the directory, you get an error.

Example of the SET DEBUG FILE TO Statement

The following example sends the output of the SET DEBUG FILE TO statement to a file called **debugging.out**:

```
SET DEBUG FILE TO 'debugging' || '.out'
```

References

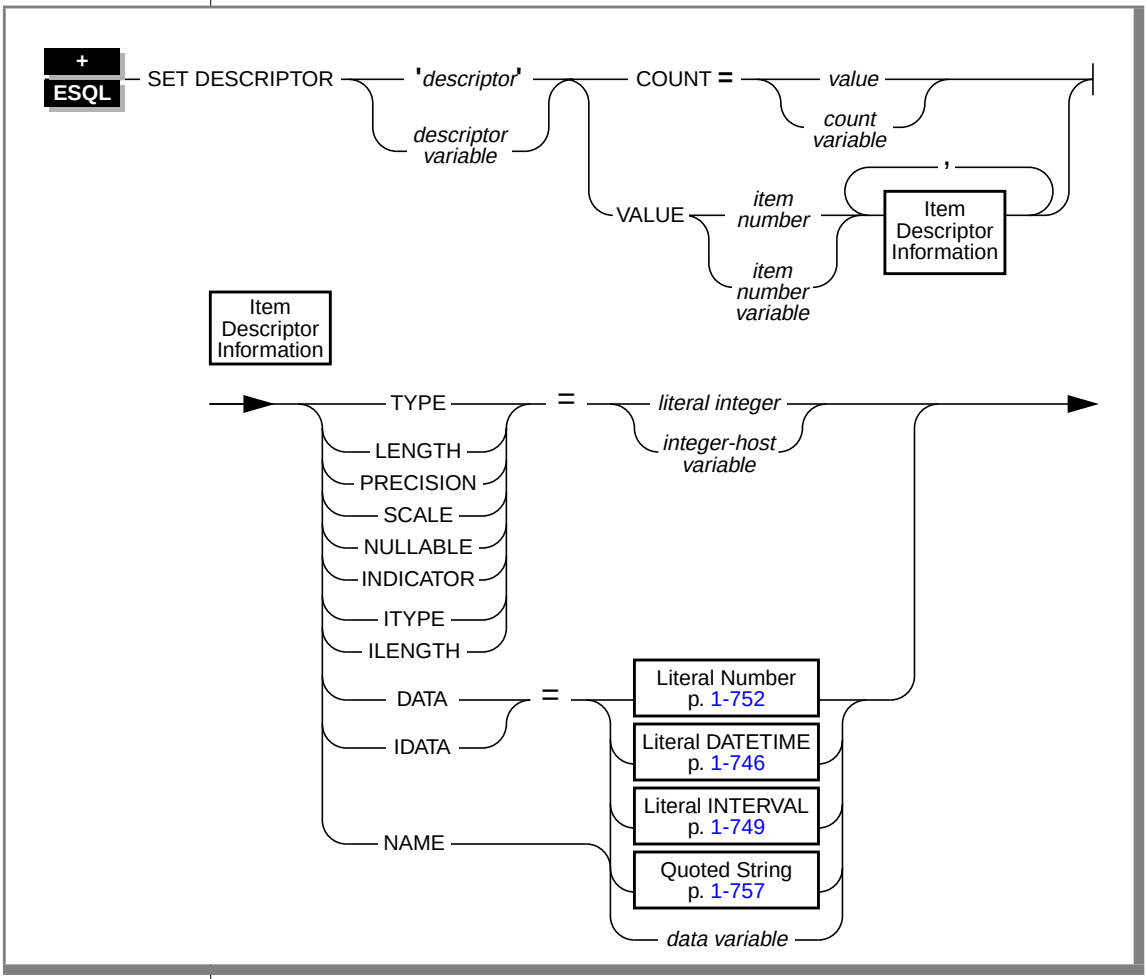
See the TRACE statement in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of stored procedures in Chapter 12.

SET DESCRIPTOR

Use the SET DESCRIPTOR statement to assign values to a system-descriptor area.

Syntax



Element	Purpose	Restrictions	Syntax
count variable	A host variable that holds a literal integer. This integer specifies how many items are actually described in the system-descriptor area.	See restriction for *value* in this table.	The name of the host variable must conform to language-specific rules for variable names.
data variable	A host variable that contains the information for the specified field (DATA, IDATA, or NAME) in the specified item descriptor	The information that is contained in *data variable* must be appropriate for the specified field.	The name of the host variable must conform to language-specific rules for variable names.
descriptor	A string that identifies the system-descriptor area to which values will be assigned	The system-descriptor area must have been previously allocated with the ALLOCATE DESCRIPTOR statement.	Quoted String, p. 1-757
descriptor variable	A host variable that holds the value of *descriptor*	The same restrictions apply to *descriptor variable* as apply to *descriptor*.	The name of the host variable must conform to language-specific rules for variable names.
integer host variable	The name of a host variable that contains the value of *literal integer*	The same restrictions apply to *integer host variable* as apply to *literal integer*.	The name of the host variable must conform to language-specific rules for variable names.
item number	An unsigned integer that specifies one of the occurrences (item descriptors) in the system-descriptor area	The value of *item number* must be greater than 0 and less than (or equal to) the number of occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 1-752

(1 of 2)

Element	Purpose	Restrictions	Syntax
item number variable	The name of an integer host variable that holds the value of *item number*	The same restrictions apply to *item number variable* as apply to *item number*.	The name of the host variable must conform to language-specific rules for variable names.
literal integer	A positive, nonzero integer that assigns a value to the specified field in the specified item descriptor. The specified field must be one of the following keywords: TYPE, LENGTH, PRECISION, SCALE, NULLABLE, INDICATOR, ITYPE, or ILENGTH.	The restrictions that apply to *literal integer* vary with the field type you specify in the VALUE option (TYPE, LENGTH, and so on). For information on the codes that are allowed for the TYPE field and their meaning, see "Setting the TYPE Field" on page 1-544. For the restrictions that apply to other field types, see the individual headings for field types under "VALUE Option" on page 1-543.	Literal Number, p. 1-752
value	A literal integer that specifies how many items are actually described in the system-descriptor area	The integer that *value* specifies must be greater than 0 and less than (or equal to) the number of occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.	Literal Number, p. 1-752

(2 of 2)

Usage

Use the SET DESCRIPTOR statement to assign values to a system-descriptor area in the following instances:

- To set the COUNT field of a system-descriptor area to match the number of items for which you are providing descriptions in the system-descriptor area (typically the items are in a WHERE clause)
- To set the item descriptor fields for each value for which you are providing descriptions in the system-descriptor area (typically the items are in a WHERE clause)

- To modify the contents of an item-descriptor field after you use the DESCRIBE statement to fill the fields for a SELECT or an INSERT statement

If an error occurs during the assignment to any identified system-descriptor fields, the contents of all identified fields are set to 0 or null, depending on the variable type.

COUNT Option

Use the COUNT option to set the number of items that are to be used in the system-descriptor area.

If you allocate a system-descriptor area with more items than you are using, you need to set the COUNT field to the number of items that you are actually using. The following example shows the sequence of statements in INFORMIX-ESQL/C that can be used in a program:

```
EXEC SQL BEGIN DECLARE SECTION;
    int count;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate descriptor 'desc_100'; /*allocates for 100 items*/

count = 2;
EXEC SQL set descriptor 'desc_100' count = :count;
```

VALUE Option

Use the VALUE option to assign values from host variables into fields for a particular item in a system-descriptor area. You can assign values for items for which you are providing a description (such as parameters in a WHERE clause), or you can modify values for items that the database server described during a DESCRIBE statement.

Setting the TYPE Field

Use the following codes to set the value of TYPE for each item.

SQL Data Type	Integer Value
CHAR	0
SMALLINT	1
INTEGER	2
FLOAT	3
SMALLFLOAT	4
DECIMAL	5
SERIAL	6
DATE	7
MONEY	8
DATETIME	10
BYTE	11
TEXT	12
VARCHAR	13
INTERVAL	14
NCHAR	15
NVARCHAR	16

E/C

For code that is easier to maintain, use the predefined constants for these SQL data types instead of their actual integer value. These constants are defined in the **sqltypes.h** header file. ♦

The following example shows how you can set the TYPE field in ESQL/C:

```
main()
{
EXEC SQL BEGIN DECLARE SECTION;
    int itemno, type;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL allocate descriptor 'desc1' with max 5;
...
EXEC SQL set descriptor 'desc1' value 2 type = 5;

type = 2; itemno = 3;
EXEC SQL set descriptor 'desc1' value :itemno type = :type;
}
```

Compiling Without the *-xopen* Option

If you do not compile using the **-xopen** option, the regular Informix <vk>SQL code is assigned for TYPE. You must be careful not to mix normal and X/Open modes because errors can result. For example, if a particular type is not defined under X/Open mode but is defined under normal mode, executing a SET DESCRIPTOR statement can result in an error.

Setting the TYPE Field in X/Open Programs

X/O

In X/Open mode, you must use the X/Open set of integer codes for the data type in the TYPE field. The following table shows the X/Open codes for data types.

SQL Data Type	Integer Value
CHAR	1
SMALLINT	4
INTEGER	5
FLOAT	6
DECIMAL	3

If you use the ILENGTH, IDATA, or ITYPE fields in a SET DESCRIPTOR statement, a warning message appears. The warning indicates that these fields are not standard X/Open fields for a system-descriptor area. ♦

E/C

For code that is easier to maintain, use the predefined constants for these X/Open SQL data types instead of their actual integer value. These constants are defined in the **sqlxtype.h** header file. ♦

Setting the DATA Field

When you set the **DATA** field, you must provide the appropriate type of data (character string for CHAR or VARCHAR, integer for INTEGER, and so on).

When any value other than DATA is set, the value of DATA is undefined. You cannot set the **DATA** field for an item without setting TYPE for that item. If you set the **TYPE** field for an item to a character type, you must also set the **LENGTH** field. If you do not set the **LENGTH** field for a character item, you receive an error.

Using LENGTH or ILENGTH

If your **DATA** or **IDATA** field contains a character string, you must specify a value for **LENGTH**. If you specify LENGTH=0, **LENGTH** sets automatically to the maximum length of the string. The **DATA** or **IDATA** field can contain a 368-literal character string or a character string derived from a character variable of CHAR or VARCHAR data type. This provides a method to determine the length of a string in the **DATA** or **IDATA** field dynamically.

If a DESCRIBE statement precedes a SET DESCRIPTOR statement, **LENGTH** automatically sets to the maximum length of the character field that is specified in your table.

This information is identical for **ILENGTH**.

Using DECIMAL or MONEY Data Types

If you set the **TYPE** field for a DECIMAL or MONEY data type, and you want to use a scale or precision other than the default values, set the **SCALE** and **PRECISION** fields. You do not need to set the **LENGTH** field for a DECIMAL or MONEY item; the **LENGTH** field is set accordingly from the **SCALE** and **PRECISION** fields.

Using DATETIME or INTERVAL Data Types

If you set the **TYPE** field for a DATETIME or INTERVAL value, you can set the **DATA** field as a literal DATETIME or INTERVAL or as a character string. If you use a character string, you must set the **LENGTH** field to the encoded qualifier value.

E/C

To determine the encoded qualifiers for a DATETIME or INTERVAL character string, use the datetime and interval macros in the **datetime.h** header file.

If you set DATA to a host variable of DATETIME or INTERVAL, you do not need to set **LENGTH** explicitly to the encoded qualifier integer. ♦

E/CO

To determine the encoded qualifiers for a DATETIME or INTERVAL character string, use the ECO-IQU routine. ♦

Setting the INDICATOR Field

If you want to put a null value into the system-descriptor area, set the **INDICATOR** field to -1, and do not set the **DATA** field.

If you set the **INDICATOR** field to 0 to indicate that the data is not null, you must set the **DATA** field.

Setting the ITYPE Field

The **ITYPE** field expects an integer constant that indicates the data type of your indicator variable. Use the same set of constants as for the **TYPE** field. The constants are listed on page 1-544.

Modifying Values Set by the DESCRIBE Statement

You can use a DESCRIBE statement to modify the contents of a system-descriptor area after it is set.

E/CO

To determine the encoded qualifiers for a DATETIME or INTERVAL character string, use the ECO-IQU routine. See the *INFORMIX-ESQL/COBOL Programmer's Manual* for information on this routine.

After you use a DESCRIBE statement on SELECT or an INSERT statement, you must check to determine whether the **TYPE** field is set to either 11 or 12 to indicate a TEXT or BYTE data type. If **TYPE** contains an 11 or a 12, you must use the SET DESCRIPTOR statement to reset **TYPE** to 116, which indicates FILE type. ♦

References

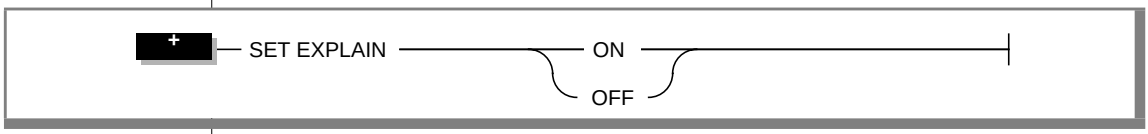
See the ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DECLARE, DESCRIBE, EXECUTE, FETCH, GET DESCRIPTOR, OPEN, PREPARE, and PUT statements in this manual for further information about using dynamic SQL statements.

For further information about the system-descriptor area, see your SQL API product manual.

SET EXPLAIN

Use the SET EXPLAIN statement to obtain a measure of the work involved in performing a query.

Syntax



Usage

The SET EXPLAIN statement executes during the database server optimization phase, which occurs when you initiate a query. For queries that are associated with a cursor, if the query is prepared and does not have host variables, optimization occurs when you prepare it; otherwise, it occurs when you open the cursor.

When you issue a SET EXPLAIN ON statement, the path that the optimizer chooses for each subsequent query is written to the **sqexplain.out** file. The SET EXPLAIN ON statement remains in effect until you issue a SET EXPLAIN OFF statement or until the program ends. The owner name (for example, *owner.customer)* qualifies table names in the **sqexplain.out** file.

If the file already exists, subsequent output is appended to the file. If the client application and the database server are on the same computer, the **sqexplain.out** file is stored in your current directory.

When the current database is on another computer, the **sqexplain.out** file is stored in your home directory on the remote host. If you do not have a home directory on the remote host, the program stores **sqexplain.out** in the directory from which the database server was started.

SE

If you do not have write privileges to a directory, INFORMIX-SE generates an error. ♦

SET EXPLAIN Output

The SET EXPLAIN output file contains a copy of the query, a plan of execution that the database-server optimizer selects, and an estimate of the amount of work. The optimizer selects a plan to provide the most efficient way to perform the query, based on such things as the presence and type of indexes and the number of rows in each table.

The optimizer uses an estimate to compare the cost of one path with another. The estimated cost does not translate directly into time. However, when data distributions are used, a query with a higher estimate generally takes longer to run than one with a smaller estimate.

The estimated cost of the query is included in the SET EXPLAIN output. In the case of a query and a subquery, two estimated cost figures are returned; the query figure also contains the subquery cost. The subquery cost is shown only so you can see the cost that is associated with the subquery.

In addition to the estimated cost, the output file contains the following information:

- An estimate of the number of rows to be returned
- The order in which tables are accessed during execution
- The table column or columns that serve as a filter, if any, and whether the filtering occurs through an index
- The method (access path) by which the executor reads each table. The following list shows the possible methods.

Method	Effect
SEQUENTIAL SCAN	Reads rows in sequence
INDEX PATH	Scans one or more indexes
AUTOINDEX PATH	Creates a temporary index
SORT SCAN	Sorts the result of the preceding join or table scan
MERGE JOIN	Uses a sort / merge join instead of nested-loop join
REMOTE PATH	Accesses another distributed database
HASH JOIN	Uses a hash join

The optimizer chooses the best path of execution to produce the fastest possible table join using a nested-loop join or sort-merge join wherever appropriate.

The SORT SCAN section indicates that sorting the result of the preceding join or table scan is necessary for a sort-merge join. It includes a list of the columns that form the sort key. The order of the columns is the order of the sort. As with indexes, the default order is *ascending*. Where possible, this ordering is arranged to support any requested ORDER BY or GROUP BY clause. If the ordering can be generated from a previous sort or an index lookup, the SORT SCAN section does not appear.

The MERGE JOIN section indicates that a sort-merge join, instead of the nested-loop join, is to be used on the preceding join / table pair. It includes a list of the filters that control the sort-merge join and, where applicable, a list of any other join filters. For example, a join of tables A and B with the filters `A.c1 = B.c1` and `A.c2 < B.c2` lists the first join under "Merge Filters" and the second join under "Other Join Filters."

The DYNAMIC HASH JOIN section indicates that a hash join is to be used on the preceding join / table pair. It includes a list of the filters used to join the tables together.

A dynamic hash join uses one of the tables to construct a *hash* index and adds the index for the other table into the hash index. This is referred to as the *build phase*. If DYNAMIC HASH JOIN is followed by the (Build Outer) in the output, then the build phase is occurring on the first table; otherwise it occurs on the second table, preceding the DYNAMIC HASH JOIN. In the following example, the build phase occurs on table **username.a**:

```
SELECT a.adatetime FROM manytypes a, alltypes b
    WHERE a.adatetime = b.adate and a.along + 7 = b.along/3

Estimated Cost: 10
Estimated # of Rows Returned: 2

1) username.a: SEQUENTIAL SCAN
2) username.b: SEQUENTIAL SCAN

DYNAMIC HASH JOIN
    Dynamic Hash Filters: username.a.adatetime =
    username.b.adate and a.along + 7 = b.along/3
```

SE

When data distributions are not used, an INFORMIX-SE database server generates fewer query-processing statistics than are available from an OnLine database server. As a result, estimates for the cost and the number of rows that are returned might be more precise if you use INFORMIX-OnLine Dynamic Server than if you use INFORMIX-SE. Estimates returned for queries that include joins tend to be highly inaccurate. ♦

The following output examples represent what you might see when a SET EXPLAIN ON statement is issued using INFORMIX-OnLine Dynamic Server.

The first two examples contain two entries for a multiple-table query and show the SORT SCAN and MERGE JOIN lines. Note that in both cases, if SORT MERGE was not chosen, the second table would have been scanned using an *autoindex path.* An autoindex path is an index constructed automatically at execution time by the database server. It is removed when the query completes.

```
QUERY:
------------
select i.stock_num from items i, stock s, manufact m
    where i.stock_num = s.stock_num
    and i.manu_code = s.manu_code
    and s.manu_code = m.manu_code

Estimated Cost: 52
Estimated # of Rows Returned: 130

1) rdtest.m: SEQUENTIAL SCAN

SORT SCAN: rdtest.m.manu_code

2) rdtest.s: SEQUENTIAL SCAN

SORT SCAN: rdtest.s.manu_code

MERGE JOIN:
   Merge Filters: rdtest.m.manu_code = rdtest.s.manu_code

3) rdtest.i: INDEX PATH

(1) Index Keys: stock_num manu_code
    Lower Index Filter: (rdtest.i.stock_num = rdtest.s.stock_num AND
    rdtest.i.manu_code = rdtest.s.manu_code)

QUERY:
------------
select stock.description from stock, stock2
    where stock.description = stock2.description
    and stock.unit_price < stock2.unit_price

Estimated Cost: 15
Estimated # of Rows Returned: 370

1) rdtest.stock: SEQUENTIAL SCAN
```

```
SORT SCAN: rdtest.stock.description

2) rdtest.stock2: SEQUENTIAL SCAN

SORT SCAN: rdtest.stock2.description

MERGE JOIN
   Merge Filters: rdtest.stock2.description = rdtest.stock.description
   Other Join Filters: rdtest.stock.unit_price < rdtest.stock2.unit_price
```

The following example shows the SET EXPLAIN output for a simple query and a complex query from the **customer** table:

```
QUERY:
------------
SELECT fname, lname, company FROM customer

Estimated Cost: 3
Estimated # of Rows Returned: 28

1) joe.customer: SEQUENTIAL SCAN


QUERY:
 ------
SELECT fname, lname, company FROM customer
    WHERE company MATCHES 'Sport*' AND customer_num BETWEEN 110 AND 115
    ORDER BY lname;

Estimated Cost: 4
Estimated # of Rows Returned: 1
Temporary Files Required For: Order By

1) joe.customer: INDEX PATH

Filters: joe.customer.company MATCHES 'Sport*'

 (1) Index Keys: customer_num
  Lower Index Filter: joe.customer.customer_num >= 110
  Upper Index Filter: joe.customer.customer_num <= 115
```

The following example shows the SET EXPLAIN output for a multiple-table query:

```
QUERY:
------------
SELECT * FROM customer, orders, items
    WHERE customer.customer_num = orders.customer_num
    AND orders.order_num = items.order_num

Estimated Cost: 20
Estimated # of Rows Returned: 69

1) joe.orders: SEQUENTIAL SCAN

2) joe.customer: INDEX PATH

 (1) Index Keys: customer_num
```

```
   Lower Index Filter: joe.customer.customer_num = joe.orders.customer_num

 3) joe.items: INDEX PATH

   (1) Index Keys: order_num
   Lower Index Filter: joe.items.order_num  = joe.orders.order_num
```

SET EXPLAIN Output with Fragmentation and PDQ

When the table is fragmented, the output shows which table or index is scanned. Fragments are identified with a fragment number. The fragment numbers are the same as those contained in the dbspace column in the **sysfragments** system catalog table. If the optimizer must scan all fragments (that is, if it is unable to eliminate any fragment from consideration), the optimizer indicates this with ALL. In addition, if the optimizer eliminates all the fragments from consideration, that is, none of the fragments contain the queried information, the optimizer indicates this with NONE. For information on how OnLine eliminates a fragment from consideration, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

When PDQ is turned on, the output shows whether the optimizer used parallel scans. If the optimizer used parallel scans, the output shows PARALLEL; if PDQ is turned off, the output shows SERIAL. If PDQ is turned on, the optimizer indicates the maximum number of threads that are required to answer the query. The output shows `# of Secondary Threads`. This field indicates the number of threads that are required in addition to your user session thread. The total number of threads necessary is the number of secondary threads plus 1.

The output indicates when a hash join is used. The query is marked with DYNAMIC HASH JOIN, and the table on which the hash is built is marked with Build Outer.

The following example shows the SET EXPLAIN output for a table with fragmentation and PDQ priority set to low:

```
select * from t1 where c1 > 20

Estimated Cost: 2
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Parallel, fragments: 2)

 Filters: informix.t1.c1 > 20

# of Secondary Threads = 1
```

The following example of SET EXPLAIN output shows a table with fragmentation but without PDQ:

```
select * from t1 where c1 > 12

Estimated Cost: 3
Estimated # of Rows Returned: 2

1) informix.t1: SEQUENTIAL SCAN (Serial, fragments: 1, 2)

    Filters: informix.t1.c1 > 12
```

The following example of SET EXPLAIN output shows a table with hash join (fragmentation, and PDQ priority set to ON). The hash join is created when you create an equality join between two tables that are not indexed.

```
QUERY:
------
select h1.c1, h2.c1 from h1, h2 where h1.c1=h2.c1

Estimated Cost: 2
Estimated # of Rows Returned: 5

1) informix.h1: SEQUENTIAL SCAN (Parallel, fragments: ALL)

2) informix.h2: SEQUENTIAL SCAN (Parallel, fragments: ALL)


DYNAMIC HASH JOIN (Build Outer)
 Dynamic Hash Filters: informix.h1.c1 = informix.h2.c1

# of Secondary Threads = 6
```

The following example of SET EXPLAIN output shows a table with fragmentation, with PDQ priority set to LOW, and an index that was selected as the search method:

```
QUERY:
------
select * from t1 where c1 < 13

Estimated Cost: 2
Estimated # of Rows Returned: 1

1) informix.t1: INDEX PATH

 (1) Index Keys: c1 (Parallel, fragments: ALL)
 Upper Index Filter: informix.t1.c1 < 13


# of Secondary Threads = 3
```

Using SET EXPLAIN With SET OPTIMIZATION

If you SET OPTIMIZATION to low, the output of SET EXPLAIN displays the following uppercase string:

```
QUERY:{LOW}
```

If you SET OPTIMIZATION to high, the output of SET EXPLAIN displays the following uppercase string:

```
QUERY:
```

Reference

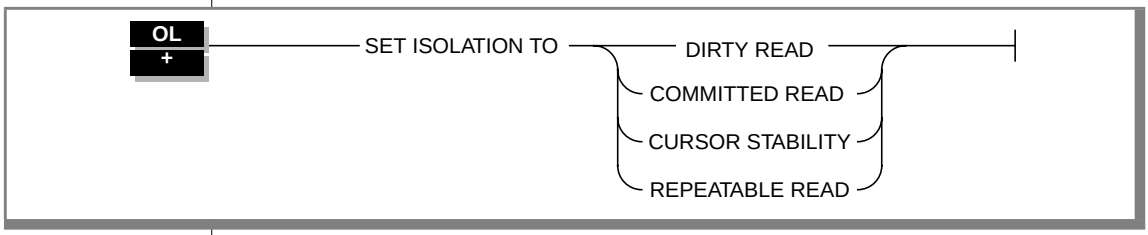
In the *INFORMIX-OnLine Dynamic Server Performance Guide*, see the discussion of SET EXPLAIN and the optimizer discussion.

SET ISOLATION

Use the SET ISOLATION statement with the INFORMIX-OnLine Dynamic Server database server to define the degree of concurrency among processes that attempt to access the same rows simultaneously.

The SET ISOLATION statement is an Informix extension to the ANSI SQL-92 standard. If you want to set isolation levels through an ANSI-compliant statement, use the SET TRANSACTION statement instead. See the SET TRANSACTION statement on page 1-575 for a comparison of these two statements.

Syntax



Usage

The database isolation level affects read concurrency when rows are retrieved from the database. INFORMIX-OnLine Dynamic Server uses shared locks to support four levels of isolation among processes attempting to access data.

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with rows that you are updating or deleting. If another process attempts to update or delete rows that you are reading with an isolation level of Repeatable Read, that process will be denied access to those rows.

ESQL

Cursors that are currently open when you execute the SET ISOLATION statement might or might not use the new isolation level when rows are later retrieved. The isolation level in effect could be any level that was set from the time the cursor was opened until the time the application actually fetches a row. The database server might have read rows into internal buffers and internal temporary tables using the isolation level that was in effect at that time. To ensure consistency and reproducible results, close open cursors before you execute the SET ISOLATION statement. ♦

Informix Isolation Levels

The following definitions explain the critical characteristics of each isolation level, from the lowest level of isolation to the highest.

Isolation Level	Characteristics
Dirty Read	Provides zero isolation. Dirty Read is appropriate for static tables that are used for queries. With a Dirty Read isolation level, a query might return a *phantom* row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. No other isolation level allows access to a phantom row. Dirty Read is the only isolation level available to databases that do not have transactions.
Committed Read	Guarantees that every retrieved row is committed in the table at the time that the row is retrieved. Even so, no locks are acquired. After one process retrieves a row because no lock is held on the row, another process can acquire an exclusive lock on the same row and modify or delete data in the row. Committed Read is the default level of isolation in a database with logging that is not ANSI compliant.

(1 of 2)

Isolation Level	Characteristics
Cursor Stability	Acquires a shared lock on the selected row. Another process can also acquire a shared lock on the same row, but no process can acquire an exclusive lock to modify data in the row. When you fetch another row or close the cursor, INFORMIX-OnLine Dynamic Server releases the shared lock. If you set the isolation level to Cursor Stability, but you are not using a transaction, the Cursor Stability isolation level acts like the Committed Read isolation level. Locks are acquired when the isolation level is set to Cursor Stability outside a transaction, but they are released immediately at the end of the statement that reads the row.
Repeatable Read	Acquires a shared lock on every row that is selected during the transaction. Another process can also acquire a shared lock on a selected row, but no other process can modify any selected row during your transaction. If you repeat the query during the transaction, you reread the same information. The shared locks are released only when the transaction commits or rolls back. Repeatable Read is the default isolation level in an ANSI-compliant database.

(2 of 2)

Default Isolation Levels

The default isolation level for a particular database is established when you create the database according to database type. The following list describes the default isolation level for each database type.

Isolation Level	Database Type
Dirty Read	Default level of isolation in a database without logging
Committed Read	Default level of isolation in a database with logging that is not ANSI compliant
Repeatable Read	Default level of isolation in an ANSI-compliant database

The default level remains in effect until you issue a SET ISOLATION statement. After a SET ISOLATION statement executes, the new isolation level remains in effect until one of the following events occurs:

- You enter another SET ISOLATION statement.
- You open another database that has a default isolation level different from the isolation level that your last SET ISOLATION statement specified.
- The program ends.

Effects of Isolation Levels

You cannot set the database isolation level in a database that does not have logging. Every retrieval in such a database occurs as a Dirty Read.

You can issue a SET ISOLATION statement from a client computer only after a database has been opened.

The data obtained during blob retrieval can vary, depending on the database isolation level. Under Dirty Read or Committed Read levels of isolation, a process is permitted to read a blob that is either deleted (if the delete is not yet committed) or in the process of being deleted. Under these isolation levels, an application can read a deleted blob when certain conditions exist. See the *INFORMIX-OnLine Dynamic Server Administrator's Guide* for information about these conditions.

DB

When you use DB-Access, you see more lock conflicts with higher levels of isolation. For example, if you use Cursor Stability, you see more lock conflicts than if you use Committed Read. ♦

ESQL

If you use a scroll cursor in a transaction, you can force consistency between your temporary table and the database table either by setting the isolation level to Repeatable Read or by locking the entire table during the transaction.

If you use a scroll cursor with hold in a transaction, you cannot force consistency between your temporary table and the database table. A table-level lock or locks that are set by Repeatable Read are released when the transaction is completed, but the scroll cursor with hold remains open beyond the end of the transaction. You can modify released rows as soon as the transaction ends, but the retrieved data in the temporary table might be inconsistent with the actual data. ♦

References

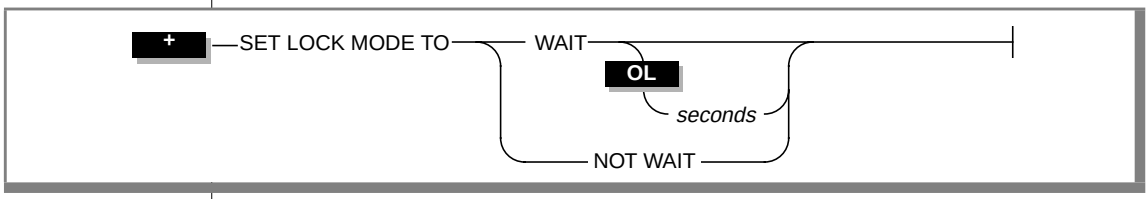
See the CREATE DATABASE, SET LOCK MODE, and SET TRANSACTION statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of isolation levels in Chapter 7.

SET LOCK MODE

Use the SET LOCK MODE statement to define how the database server handles a process that tries to access a locked row or table.

Syntax



Element	Purpose	Restrictions	Syntax
seconds	The maximum number of seconds that a process waits for a lock to be released. If the lock is still held at the end of the waiting period, the database server ends the operation and returns an error code to the process.	In a networked environment, the DBA establishes a default value for the waiting period by using the ONCONFIG parameter DEADLOCK_TIMEOUT. See "WAIT Keyword" on page 1-562 for an explanation of when the *seconds* parameter overrides the DEADLOCK_TIMEOUT parameter.	Literal Number, p. 1-752

Usage

You can direct the response of the database server in the following ways when a process tries to access a locked row or table.

Lock Mode	Effect
NOT WAIT	Ends the operation immediately and returns an error code. This condition is the default.
WAIT	Suspends the process until the lock releases
WAIT *seconds*	Suspends the process until the lock releases or until the end of a waiting period, which is specified in seconds. If the lock remains after the waiting period, it ends the operation and returns an error code.

SE

INFORMIX-SE does not support the *seconds* parameter. If you decide that a process should wait for a lock to release, you cannot limit the waiting period.

The SET LOCK MODE statement is available on computers that use kernel locking. To determine whether your computer uses kernel locking, check the directory that holds the database files. If the directory contains files with the extension **.lok**, your system does not use kernel locking, and the SET LOCK MODE statement is unavailable. ♦

WAIT Keyword

The database server protects against the possibility of a deadlock when you request the WAIT option. Before the database server suspends a process, it checks whether suspending the process could create a deadlock. If the database server discovers that a deadlock could occur, it ends the operation (overruling your instruction to wait) and returns an error code. In the case of either a suspected or actual deadlock, the database server returns an error.

Cautiously use the unlimited waiting period that was created when you specify the WAIT option without *seconds*. If you do not specify an upper limit, and the process that placed the lock somehow fails to release it, suspended processes could wait indefinitely. Because a true deadlock situation does not exist, the database server does not take corrective action.

In a networked environment, the DBA uses the ONCONFIG parameter DEADLOCK_TIMEOUT to establish a default value for *seconds*. If you use a SET LOCK MODE statement to set an upper limit, your value applies only when your waiting period is shorter than the system default. The number of seconds that the process waits applies only if you acquire locks within the current database server and a remote database server within the same transaction.

References

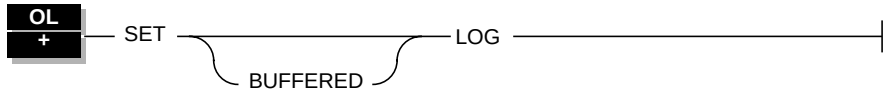
See the LOCK TABLE, UNLOCK TABLE, SET ISOLATION, and SET TRANSACTION statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of SET LOCK MODE in Chapter 7.

SET LOG

Use the SET LOG statement to change your INFORMIX-OnLine Dynamic Server database logging mode from buffered transaction logging to unbuffered transaction logging or vice versa.

Syntax



Usage

You activate transaction logging when you create a database or add logging to an existing database. These transaction logs can be buffered or unbuffered.

The default condition for transaction logs is unbuffered logging. As soon as a transaction ends, the OnLine database server writes the transaction to the disk. If a system failure occurs when you are using unbuffered logging, you recover all completed transactions.

You gain a marginal increase in efficiency with buffered logging, but you incur some risk. In the event of a system failure, the OnLine database server cannot recover the completed transactions that were buffered in memory.

The SET LOG statement changes the transaction-logging mode to unbuffered logging; the SET BUFFERED LOG statement changes the mode to buffered logging.

The SET LOG statement redefines the mode for the current session only. The default mode, which the OnLine administrator sets using ON-Monitor, remains unchanged.

The buffering option does not affect retrievals from external tables. For distributed queries, a database with logging can retrieve only from databases with logging, but it makes no difference whether the databases use buffered or unbuffered logging.

ANSI

An ANSI-compliant database cannot use buffered logs. ♦

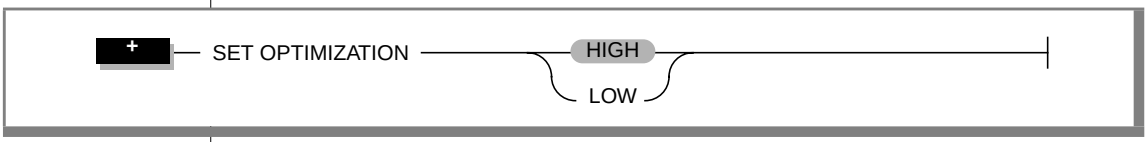
References

See the CREATE DATABASE and START DATABASE statements in this manual.

SET OPTIMIZATION

Use the SET OPTIMIZATION statement to specify a high or low level of database-server optimization.

Syntax



Usage

You can execute a SET OPTIMIZATION statement at any time. The optimization level carries across databases but applies only within the current database server.

After a SET OPTIMIZATION statement executes, the new optimization level remains in effect until you enter another SET OPTIMIZATION statement or until the program ends.

The default database server optimization level, HIGH, remains in effect until you issue another SET OPTIMIZATION statement. The LOW option invokes a less sophisticated, but faster, optimization algorithm.

The algorithm that a SET OPTIMIZATION HIGH statement invokes is a sophisticated, cost-based strategy that examines all reasonable choices and selects the best overall alternative. For large joins, this algorithm can incur more overhead than desired. In extreme cases, you can run out of memory.

The alternative algorithm that a SET OPTIMIZATION LOW statement invokes eliminates unlikely join strategies during the early stages, which reduces the time and resources spent during optimization. However, when you specify a low level of optimization, the optimal strategy might not be selected because it was eliminated from consideration during early stages of the algorithm.

The following example shows optimization across a network. The **central** database (on computer 1) is to have LOW optimization; the **western** database (on computer 2) is to have HIGH optimization. If the **western** database were on the same computer as **central**, it would have LOW optimization.

```
CONNECT TO 'central';
SET OPTIMIZATION low;
SELECT catalog.*, description, unit_price, unit,
       unit_descr, manu_name, lead_time
    FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
CLOSE DATABASE;
CONNECT TO 'western@rockie';
SET OPTIMIZATION low;
SELECT catalog.*, description, unit_price, unit,
       unit_descr, manu_name, lead_time
    FROM catalog, stock, manufact
    WHERE catalog.stock_num = stock.stock_num
        AND stock.manu_code = manufact.manu_code
        AND catalog_num = 10025
```

Optimizing Stored Procedures

For stored procedures that remain unchanged or change only slightly, you might want to set the SET OPTIMIZATION statement to HIGH when you create the procedure. This stores the best query plans for the procedure. Then SET OPTIMIZATION to LOW before you execute the procedure. The procedure then uses the optimal query plans and runs at the more cost-effective rate.

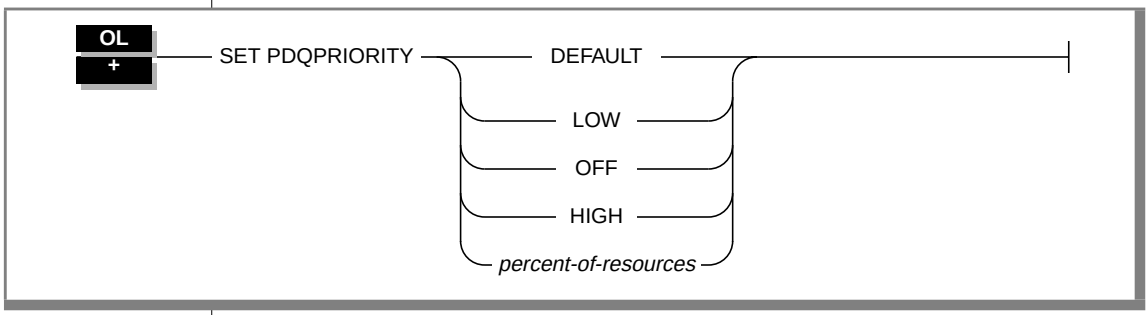
References

In the *INFORMIX-OnLine Dynamic Server Performance Guide*, see the discussion of optimizing queries.

SET PDQPRIORITY

The SET PDQPRIORITY statement allows an application to set the query priority level dynamically within an application.

Syntax



Element	Purpose	Restrictions	Syntax
percent-of-resources	An integer value that specifies the query priority level and the amount of resources the database server uses in order to process the query	You must specify a value in the following range: -1, 0, 1 to 100. The values -1, 0, and 1 have special meanings. See "Meaning of SET PDQPRIORITY Parameters" on page 1-569 for an explanation of these values.	Literal Number, p. 1-752

Usage

Priority set with the SET PDQPRIORITY statement overrides the environment variable **PDQPRIORITY**. However, no matter what priority value you set with the SET PDQPRIORITY statement, the ONCONFIG configuration parameter MAX_PDQPRIORITY determines the actual priority value that the INFORMIX-OnLine Dynamic Server uses for your queries.

For example, assume that the DBA has set the MAX_PDQPRIORITY parameter to 50. A user enters the following SET PDQPRIORITY statement to set the query priority level to 80.

```
SET PDQPRIORITY 80
```

When it processes the user's query, OnLine uses the value of the MAX_PDQPRIORITY parameter to factor the query priority level set by the user. OnLine silently processes the query with a priority level of 40. This priority level represents 50 percent of the 80 percent of resources specified by the user.

Meaning of SET PDQPRIORITY Parameters

The parameters that the SET PDQPRIORITY statement can use are shown in the following table.

Parameter	Meaning
DEFAULT	Uses the value that is specified in the PDQPRIORITY environment variable, if any. DEFAULT is the symbolic equivalent of -1.
LOW	Signifies that data is fetched from fragmented tables in parallel. OnLine uses no other forms of parallelism. LOW is the symbolic equivalent of 1.
OFF	Indicates that PDQ is turned off. OnLine uses no parallelism. OFF is the symbolic equivalent of 0. OFF is the default setting if you do not specify the PDQPRIORITY environment variable or the SET PDQPRIORITY statement.
HIGH	Signifies that the database server determines an appropriate value to use for PDQPRIORITY. This decision is based on several things, including the number of available processors, the fragmentation of the tables being queried, the complexity of the query, and so on. Informix reserves the right to change the performance behavior of queries when HIGH is specified in future releases.
percent-of-resources	Indicates a query priority level and indicates the percent of resources a database server uses in order to answer the query. Resources include the amount of memory and the number of processors. The higher the number, the more resources the database server uses. Although usually the more resources a database server uses indicates better performance for a given query, using too many resources can cause contention among the resources and remove resources from other queries, which results in degraded performance. Range = -1, 0, 1 to 100.

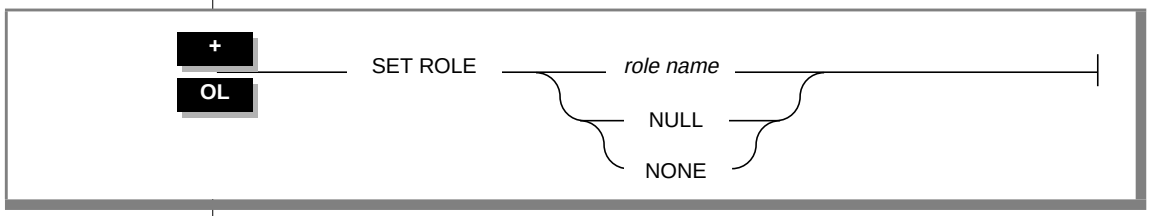
References

For information about the **PDQPRIORITY** environment variable, see the *Informix Guide to SQL: Reference*. See the *INFORMIX-OnLine Dynamic Server Administrator's Guide* for information about the ONCONFIG parameter MAX_PDQPRIORITY.

SET ROLE

Use the SET ROLE statement to enable the privileges of a role.

Syntax



Element	Purpose	Restrictions	Syntax
role name	Name of the role that you want to enable	The role must have been created with the CREATE ROLE statement.	Identifier, p. 1-723

Usage

Any user who is granted a role can enable the role using the SET ROLE statement. A user can only enable one role at a time. If a user executes the SET ROLE statement after a role is already set, the new role replaces the old role.

All users are, by default, assigned the role NULL or NONE (NULL and NONE are synonymous). The roles NULL and NONE have no privileges. When you set the role to NULL or NONE, you disable the current role.

When a user sets a role, the user gains the privileges of the role, in addition to the privileges of PUBLIC and the user's own privileges. If a role is granted to another role, the user gains the privileges of both roles, in addition to those of PUBLIC and the user's own privileges. After a SET ROLE statement executes successfully, the role remains effective until the current database is closed or the user executes another SET ROLE statement. Additionally, the user, not the role, retains ownership of all the objects, such as tables, that were created during a session.

A user cannot execute the SET ROLE statement while in a transaction. If the SET ROLE statement is executed while a transaction is active, an error occurs.

If the SET ROLE statement is executed as a part of a trigger or stored procedure, and the owner of the trigger or stored procedure was granted the role with the WITH GRANT OPTION, the role is enabled even if the user is not granted the role.

The following example sets the role **engineer:**

```
SET ROLE engineer
```

The following example sets a role and then relinquishes the role after it performs a SELECT operation:

```
EXEC SQL set role engineer;
EXEC SQL select fname, lname, project
        into :efname, :elname, :eproject
        where project_num > 100 and lname = 'Larkin';
printf ("%s is working on %s\n", efname, eproject);
EXEC SQL set role null;
```

References

See the CREATE ROLE, DROP ROLE, GRANT, and REVOKE statements in this manual.

SET SESSION AUTHORIZATION

The SET SESSION AUTHORIZATION statement lets you change the user name under which database operations are performed in the current OnLine session. This statement is enabled by the DBA privilege, which you must obtain from the DBA before the start of your current session. The new identity remains in effect in the current database until you execute another SET SESSION AUTHORIZATION statement or until you close the current database.

Syntax

Element	Purpose	Restrictions	Syntax
' *user* '	The user name under which database operations are to be performed in the current session	You must specify a valid user name. You must put quotation marks around the user name.	Identifier, p. 1-723

Usage

The SET SESSION AUTHORIZATION statement allows a user with the DBA privilege to bypass the privileges that protect database objects. You can use this statement to gain access to a table and adopt the identity of a table owner to grant access privileges. You must obtain the DBA privilege before you start a session in which you use this statement. Otherwise, this statement returns an error.

When you use this statement, the user name to which the authorization is set must have the Connect privilege on the current database. Additionally, the DBA cannot set the authorization to PUBLIC or to any defined role in the current database.

Setting a session to another user causes a change in a user name in the current active database server. In other words, these users are, as far as this database server process is concerned, completely dispossessed of any privileges that they might have while accessing the database server through some administrative utility. Additionally, the new session user is not able to initiate an administrative operation (execute a utility, for example) by virtue of the acquired identity.

After the SET SESSION AUTHORIZATION statement successfully executes, the user must use the SET ROLE statement to assume a role granted to the current user. Any role enabled by a previous user is relinquished.

Using SET SESSION AUTHORIZATION to Obtain Privileges

You can use the SET SESSION AUTHORIZATION statement either to obtain access to the data directly or to grant the database-level or table-level privileges needed for the database operation to proceed. The following example shows how to use the SET SESSION AUTHORIZATION statement to obtain table-level privileges:

```
SET SESSION AUTHORIZATION TO 'cathl';
GRANT ALL ON spec TO mary;
SET SESSION AUTHORIZATION TO 'mary';
UPDATE case
    SET col1 = SELECT state FROM zip
                WHERE zip_code = 94433;
```

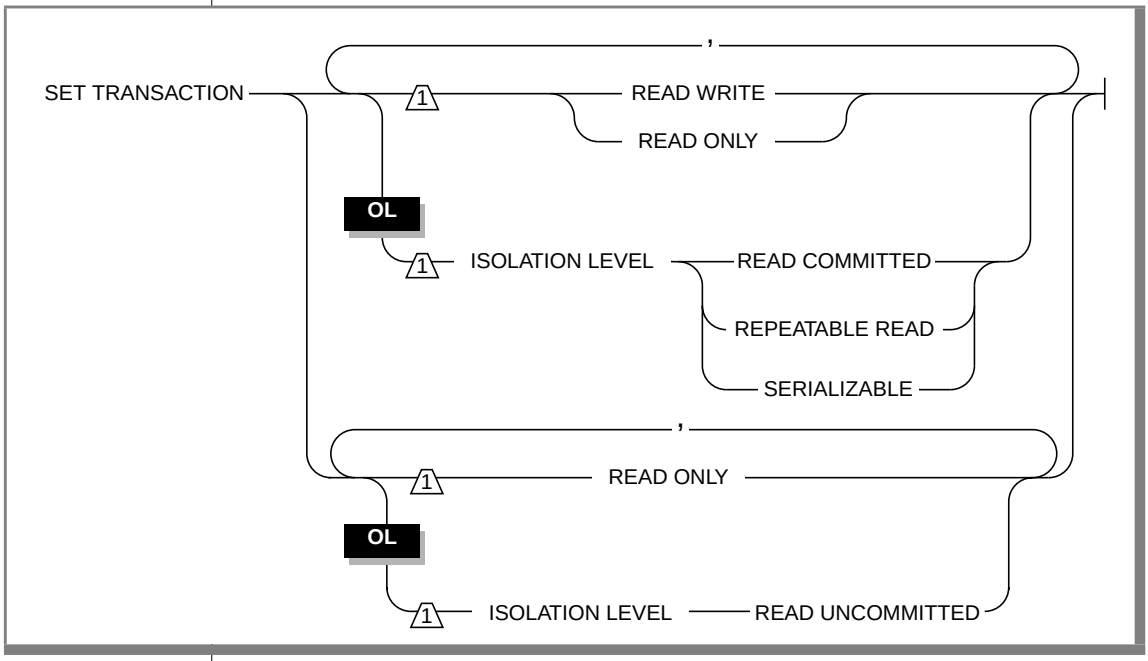
References

See the CONNECT, DATABASE, GRANT, and SET ROLE statements in this manual.

SET TRANSACTION

Use the SET TRANSACTION statement to define isolation levels and to define the access mode of a transaction (read-only or read-write).

Syntax



Usage

You can use SET TRANSACTION only in databases with logging.

You can issue a SET TRANSACTION statement from a client computer only after a database has been opened.

The database isolation level affects concurrency among processes that attempt to access the same rows simultaneously from the database. INFORMIX-OnLine Dynamic Server uses shared locks to support four levels of isolation among processes that are attempting to read data as the following list shows:

- Read Uncommitted
- Read Committed
- (ANSI) Repeatable Read
- Serializable

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with rows that you are updating or deleting; however, the access mode does affect whether you can update or delete rows. If another process attempts to update or delete rows that you are reading with an isolation level of Serializable or (ANSI) Repeatable Read, that process will be denied access to those rows.

Comparing SET TRANSACTION with SET ISOLATION

The SET TRANSACTION statement complies with ANSI SQL-92. This statement is similar to the Informix SET ISOLATION statement; however, the SET ISOLATION statement is not ANSI compliant and does not provide access modes. In fact, the isolation levels that you can set with the SET TRANSACTION statement are almost parallel to the isolation levels that you can set with the SET ISOLATION statement, as the following table shows.

SET TRANSACTION	Correlates to	SET ISOLATION
Read Uncommitted		Dirty Read
Read Committed		Committed Read
Not supported		Cursor Stability
(ANSI) Repeatable Read		(Informix) Repeatable Read
Serializable		(Informix) Repeatable Read

Another difference between the SET TRANSACTION and SET ISOLATION statements is the behavior of the isolation levels within transactions. The SET TRANSACTION statement can be issued only once for a transaction. Any cursors that are opened during that transaction are guaranteed to get that isolation level (or access mode if you are defining an access mode). With the SET ISOLATION statement, after a transaction is started, you can change the isolation level more than once within the transaction. The following examples show the SET ISOLATION and SET TRANSACTION statements, respectively:

SET ISOLATION

```
EXEC SQL BEGIN WORK;
EXEC SQL SET ISOLATION TO DIRTY READ;
EXEC SQL SELECT ... ;
EXEC SQL SET ISOLATION TO REPEATABLE READ;
EXEC SQL INSERT ... ;
EXEC SQL COMMIT WORK;
    -- Executes without error
```

SET TRANSACTION

```
EXEC SQL BEGIN WORK;
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
EXEC SQL SELECT ... ;
EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Error 876: Cannot issue SET TRANSACTION in an active
transaction.
```

Isolation Levels

The following definitions explain the critical characteristics of each isolation level, from the lowest level of isolation to the highest.

Isolation Level	Characteristics
Read Uncommitted	Provides zero isolation. Read Uncommitted is appropriate for static tables that are used for queries. With a Read Uncommitted isolation level, a query might return a *phantom* row, which is an uncommitted row that was inserted or modified within a transaction that has subsequently rolled back. Read Uncommitted is the only isolation level that is available to databases that do not have transactions.
Read Committed	Guarantees that every retrieved row is committed in the table at the time that the row is retrieved. Even so, no locks are acquired. After one process retrieves a row because no lock is held on the row, another process can acquire an exclusive lock on the same row and modify or delete data in the row. Read Committed is the default isolation level in a database with logging that is not ANSI compliant.
(ANSI) Repeatable Read	The Informix implementation of ANSI Repeatable Read. Informix uses the same approach to implement Repeatable Read that it uses for Serializable. Thus Repeatable Read meets the SQL-92 requirements.
Serializable	Acquires a shared lock on every row that is selected during the transaction. Another process can also acquire a shared lock on a selected row, but no other process can modify any selected row during your transaction. If you repeat the query during the transaction, you reread the same information. The shared locks are released only when the transaction commits or rolls back. Serializable is the default isolation level in an ANSI-compliant database.

Default Isolation Levels

The default isolation level for a particular database is established according to database type when you create the database. The default isolation level for each database type is described in the following table.

Informix	ANSI	Description
Dirty Read	Read Uncommitted	Default level of isolation in a database without logging
Committed Read	Read Committed	Default level of isolation in a database with logging that is not ANSI compliant
Repeatable Read	Serializable	Default level of isolation in an ANSI-compliant database

The default isolation level remains in effect until you issue a SET TRANSACTION statement within a transaction. After a COMMIT WORK statement completes the transaction or a ROLLBACK WORK statement cancels the transaction, the isolation level is reset to the default.

Access Modes

Both INFORMIX-OnLine Dynamic Server and INFORMIX-SE support access modes. Access modes affect read and write concurrency for rows within transactions. Use access modes to control data modification.

You can specify that a transaction is read-only or read-write through the SET TRANSACTION statement. By default, transactions are read-write. When you specify that a transaction is read-only, certain limitations apply. Read-only transactions cannot perform the following actions:

- Insert, delete, or update table rows
- Create, alter, or drop any database object such as schemas, tables, temporary tables, indexes, or stored procedures
- Grant or revoke privileges
- Update statistics
- Rename columns or tables

You can execute stored procedures in a read-only transaction as long as the procedure does not try to perform any restricted statement.

Effects of Isolation Levels

You cannot set the database isolation level in a database that does not have logging. Every retrieval in such a database occurs as a Read Uncommitted.

The data that is obtained during blob retrieval can vary, depending on the database isolation levels. Under Read Uncommitted or Read Committed isolation levels, a process is permitted to read a blob that is either deleted (if the delete is not yet committed) or in the process of being deleted. Under these isolation levels, an application can read a deleted blob when certain conditions exist. See the *INFORMIX-OnLine Dynamic Server Administrator's Guide* for information about these conditions.

ESQL

If you use a scroll cursor in a transaction, you can force consistency between your temporary table and the database table either by setting the isolation level to Serializable or by locking the entire table during the transaction.

If you use a scroll cursor with hold in a transaction, you cannot force consistency between your temporary table and the database table. A table-level lock or locks set by Serializable are released when the transaction is completed, but the scroll cursor with hold remains open beyond the end of the transaction. You can modify released rows as soon as the transaction ends, so the retrieved data in the temporary table might be inconsistent with the actual data. ♦

References

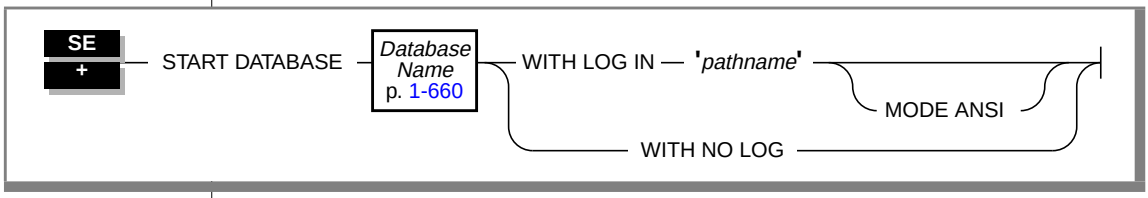
See the CREATE DATABASE, SET ISOLATION, and SET LOCK MODE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of isolation levels and concurrency issues in Chapter 7.

START DATABASE

Use the START DATABASE statement with an INFORMIX-SE database server to start recording transactions, to make a database ANSI compliant, to change the name of an existing transaction-log file, or to remove logging on a database.

Syntax



Element	Purpose	Restrictions	Syntax
pathname	The pathname and filename of the transaction log file. The default directory is the current directory.	You cannot specify an existing file in *pathname*. You must specify an existing directory in *pathname*. For maximum protection, you should specify a location for the transaction-log file that is not on the same storage device as the database.	The pathname and filename must conform to the conventions of your operating system.

Usage

To use the START DATABASE statement, all the following conditions must be true:

- You have the DBA privilege.
- No current database exists.
- The directory that is specified in *pathname* exists.

Issue a CLOSE DATABASE statement before you create and start a transaction log. The START DATABASE statement locks the database exclusively to prevent access by other processes. If another process is using the database (even if the database is only being read), the START DATABASE statement fails.

The database remains locked after the START DATABASE statement executes. When you are satisfied that the database is ready to use, execute the CLOSE DATABASE statement to remove the exclusive lock. Reopen the database with the DATABASE statement.

MODE ANSI Keyword

ANSI

Use the MODE ANSI keyword to make a database ANSI compliant. An ANSI-compliant database conforms to different transaction-processing and object-naming conventions than does a database that is not ANSI compliant.

The following example starts an ANSI-compliant database that is named **stores7**:

```
START DATABASE stores7
    WITH LOG IN '/u/myname/stores7.log' MODE ANSI
```

♦

Transaction Log Name Change

You must issue a START DATABASE statement immediately before you back up the database if you plan to change the name or the location of the transaction log. Specify the new path to the transaction log in the START DATABASE statement.

Stopping Logging

If you issue the START DATABASE statement with the WITH NO LOG clause against a database that has logging, logging is turned off after the statement is run. If you run the statement against a database that does not have logging, no error is returned. This statement cannot be run on an ANSI-compliant database.

References

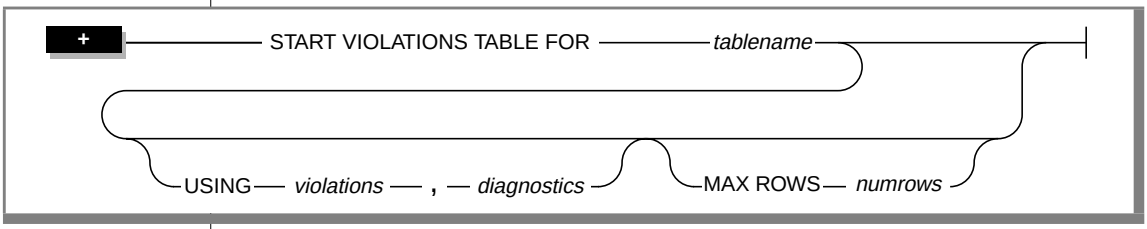
See the CREATE DATABASE and ROLLFORWARD DATABASE statements in this manual.

In the *Informix Guide to SQL: Tutorial*, see the discussion of transaction processing in Chapter 4 and Chapter 6.

START VIOLATIONS TABLE

The START VIOLATIONS TABLE statement creates a violations table and a diagnostics table for a specified target table. The database server associates the violations and diagnostics tables with the target table by recording the relationship among the three tables in the **sysviolations** system catalog table.

Syntax



Element	Purpose	Restrictions	Syntax
diagnostics	The name of the diagnostics table to be associated with the target table. The default name is the name of the target table followed by the characters **_dia**. For further information on the diagnostics table, see "Structure of the Diagnostics Table" on page 1-595.	Whether you specify the name of the diagnostics table explicitly, or the database server generates the name implicitly, the name cannot match the name of any existing table in the database.	Identifier, p. 1-723
numrows	The maximum number of rows that can be inserted into the diagnostics table when a single statement (for example, INSERT or SET) is executed on the target table. There is no default value for *numrows*. If you do not specify a value for *numrows*, there is no upper limit on the number of rows that can be inserted into the diagnostics table when a single statement is executed on the target table.	You must specify an integer value in the range 1 to the maximum value of the INTEGER data type.	Literal Number, p. 1-752

(1 of 2)

Element	Purpose	Restrictions	Syntax
table name	The name of the target table for which a violations table and diagnostics table are to be created. There is no default value.	If you do not include the USING clause in the statement, the name of the target table must be less than 15 characters. The target table cannot have a violations and diagnostics table associated with it before you execute the statement. The target table cannot be a system catalog table. The target table must be a local table.	Identifier, p. 1-723
violations	The name of the violations table to be associated with the target table. The default name is the name of the target table followed by the characters **_vio**. For further information on the violations table, see "Structure of the Violations Table" on page 1-586.	Whether you specify the name of the violations table explicitly, or the database server generates the name implicitly, the name cannot match the name of any existing table in the database.	Identifier, p. 1-723

(2 of 2)

Usage

The START VIOLATIONS TABLE statement creates the special violations table that holds rows that fail to satisfy constraints and unique indexes during insert, update, and delete operations on target tables. This statement also creates the special diagnostics table that contains information about the integrity violations caused by each row in the violations table.

Relationship of START VIOLATIONS TABLE and SET Statements

The START VIOLATIONS TABLE statement is closely related to the SET statement. If you use the SET statement to set the constraints or unique indexes defined on a table to the filtering object mode, but you do not use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for this target table, any rows that violate a constraint or unique-index requirement during an insert, update, or delete operation are not filtered out to a violations table. Instead you receive an error message indicating that you must start a violations table for the target table.

Similarly, if you use the SET statement to set a disabled constraint or disabled unique index to the enabled or filtering object mode, but you do not use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for the table on which the objects are defined, any existing rows in the table that do not satisfy the constraint or unique-index requirement are not filtered out to a violations table. If, in these cases, you want the ability to identify existing rows that do not satisfy the constraint or unique-index requirement, you must issue the START VIOLATIONS TABLE statement to start the violations and diagnostics tables before you issue the SET statement to set the objects to the enabled or filtering object mode.

Starting and Stopping the Violations and Diagnostics Tables

After you use a START VIOLATIONS TABLE statement to create an association between a target table and the violations and diagnostics tables, the only way to drop the association between the target table and the violations and diagnostics tables is to issue a STOP VIOLATIONS TABLE statement for the target table. For further information see the STOP VIOLATIONS TABLE statement on page 1-602.

Examples of START VIOLATIONS TABLE Statements

The following examples show different ways to execute the START VIOLATIONS TABLE statement.

Starting Violations and Diagnostics Tables Without Specifying Their Names

The following statement starts violations and diagnostics tables for the target table named **cust_subset**. The violations table is named **cust_subset_vio** by default, and the diagnostics table is named **cust_subset_dia** by default.

```
START VIOLATIONS TABLE FOR cust_subset
```

Starting Violations and Diagnostics Tables and Specifying Their Names

The following statement starts a violations and diagnostics table for the target table named **items**. The USING clause assigns explicit names to the violations and diagnostics tables. The violations table is to be named **exceptions**, and the diagnostics table is to be named **reasons**.

```
START VIOLATIONS TABLE FOR items
USING exceptions, reasons
```

Specifying the Maximum Number of Rows in the Diagnostics Table

The following statement starts violations and diagnostics tables for the target table named **orders**. The MAX ROWS clause specifies the maximum number of rows that can be inserted into the diagnostics table when a single statement, such as an INSERT or SET statement, is executed on the target table.

```
START VIOLATIONS TABLE FOR orders MAX ROWS 50000
```

Privileges Required for Starting Violations Tables

To start a violations and diagnostics table for a target table, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the target table and have the Resource privilege on the database.
- You must have the Alter privilege on the target table and the Resource privilege on the database.

Structure of the Violations Table

When you issue a START VIOLATIONS TABLE statement for a target table, the violations table that the statement creates has a predefined structure. This structure consists of the columns of the target table and three additional columns.

The following table shows the structure of the violations table.

Column Name	Type	Explanation
All columns of the target table, in the same order that they appear in the target table	These columns of the violations table match the data type of the corresponding columns in the target table, except that SERIAL columns in the target table are converted to INTEGER data types in the violations table.	The table definition of the target table is reproduced in the violations table so that rows that violate constraints or unique-index requirements during insert, update, and delete operations can be filtered to the violations table. Users can examine these bad rows in the violations table, analyze the related rows that contain diagnostics information in the diagnostics table, and take corrective actions.
informix_tupleid	SERIAL	This column contains the unique serial identifier that is assigned to the nonconforming row.
informix_optype	CHAR(1)	This column indicates the type of operation that caused this bad row. This column can have the following values: I = Insert D = Delete O = Update (with this row containing the original values) N = Update (with this row containing the new values) S = SET statement
informix_recowner	CHAR(8)	This column identifies the user who issued the statement that created this bad row.

Relationship Between the Violations and Diagnostics Tables

Users can take advantage of the relationships among the target table, violations table, and diagnostics table to obtain complete diagnostic information about rows that have caused data-integrity violations during INSERT, DELETE, and UPDATE statements.

Each row of the violations table has at least one corresponding row in the diagnostics table. The row in the violations table contains a copy of the row in the target table for which a data-integrity violation was detected. The row in the diagnostics table contains information about the nature of the data-integrity violation caused by the bad row in the violations table. The row in the violations table has a unique serial identifier in the **informix_tupleid** column. The row in the diagnostics table has the same serial identifier in its **informix_tupleid** column.

A given row in the violations table can have more than one corresponding row in the diagnostics table. The multiple rows in the diagnostics table all have the same serial identifier in their **informix_tupleid** column so that they are all linked to the same row in the violations table. Multiple rows can exist in the diagnostics table for the same row in the violations table because a bad row in the violations table can cause more than one data-integrity violation.

For example, a bad row can violate a unique-index requirement for one column, a not null constraint for another column, and a check constraint for yet another column. In this case, the diagnostics table contains three rows for the single bad row in the violations table. Each of these diagnostic rows identifies a different data-integrity violation that the nonconforming row in the violations table caused.

By joining the violations and diagnostics tables, the DBA or target table owner can obtain complete diagnostic information about any or all bad rows in the violations table. You can use SELECT statements to perform these joins interactively, or you can write a program to perform them within transactions.

Initial Privileges on the Violations Table

When you issue the START VIOLATIONS TABLE statement to create the violations table, the database server uses the set of privileges granted on the target table as a basis for granting privileges on the violations table. However, the database server follows different rules when it grants each type of privilege.

The following table shows the initial set of privileges on the violations table. The **Privilege** column lists the privilege. The **Condition** column explains the conditions under which the database server grants the privilege to a user.

Privilege	Condition
Insert	The user has the Insert privilege on the violations table if the user has any of the following privileges on the target table: the Insert privilege, the Delete privilege, or the Update privilege on any column.
Delete	The user has the Delete privilege on the violations table if the user has any of the following privileges on the target table: the Insert privilege, the Delete privilege, or the Update privilege on any column.
Select	The user has the Select privilege on the **informix_tupleid**, **informix_optype**, and **informix_recowner** columns of the violations table if the user has the Select privilege on any column of the target table. The user has the Select privilege on any other column of the violations table if the user has the Select privilege on the same column in the target table.
Update	The user has the Update privilege on the **informix_tupleid**, **informix_optype**, and **informix_recowner** columns of the violations table if the user has the Update privilege on any column of the target table. The user has the Update privilege on any other column of the violations table if the user has the Update privilege on the same column in the target table.

(1 of 2)

Privilege	Condition
Index	The user has the Index privilege on the violations table if the user has the Index privilege on the target table.
Alter	The Alter privilege is not granted on the violations table. (Users cannot alter violations tables.)
References	The References privilege is not granted on the violations table. (Users cannot add referential constraints to violations tables.)

(2 of 2)

The following rules apply to ownership of the violations table and privileges on the violations table:

- When the violations table is created, the owner of the target table becomes the owner of the violations table.
- The owner of the violations table automatically receives all table-level privileges on the violations table, including the Alter and References privileges. However, the database server prevents the owner of the violations table from altering the violations table or adding a referential constraint to the violations table.
- You can use the GRANT and REVOKE statements to modify the initial set of privileges on the violations table.
- When you issue an INSERT, DELETE, or UPDATE statement on a target table that has a filtering-mode unique index or constraint defined on it, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the INSERT, DELETE, or UPDATE statement on the target table provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the INSERT, DELETE, or UPDATE statement.

Similarly, when you issue a SET statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET statement provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the SET statement.

- The grantor of the initial set of privileges on the violations table is the same as the grantor of the privileges on the target table. For example, if the user **henry** has been granted the Insert privilege on the target table by both the user **jill** and the user **albert**, the Insert privilege on the violations table is granted to user **henry** both by user **jill** and by user **albert**.
- Once a violations table has been started for a target table, revoking a privilege on the target table from a user does not automatically revoke the same privilege on the violations table from that user. Instead you must explicitly revoke the privilege on the violations table from the user.
- If you have fragment-level privileges on the target table, you have the corresponding fragment-level privileges on the violations table.

Example of Privileges on the Violations Table

The following example illustrates how the initial set of privileges on a violations table is derived from the current set of privileges on the target table.

For example, assume that we have created a table named **cust_subset** and that this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives).

The following set of privileges exists on the **cust_subset** table:

- User **alvin** is the owner of the table.
- User **barbara** has the Insert and Index privileges on the table. She also has the Select privilege on the **ssn** and **lname** columns.
- User **carrie** has the Update privilege on the **city** column. She also has the Select privilege on the **ssn** column.
- User **danny** has the Alter privilege on the table.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table, as follows:

```
START VIOLATIONS TABLE FOR cust_subset
    USING cust_subset_viols, cust_subset_diags
```

The database server grants the following set of initial privileges on the **cust_subset_viols** violations table:

- User **alvin** is the owner of the violations table, so he has all table-level privileges on the table.
- User **barbara** has the Insert, Delete, and Index privileges on the violations table. She also has the Select privilege on the following columns of the violations table: the **ssn** column, the **lname** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column.
- User **carrie** has the Insert and Delete privileges on the violations table. She has the Update privilege on the following columns of the violations table: the **city** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column. She has the Select privilege on the following columns of the violations table: the **ssn** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column.
- User **danny** has no privileges on the violations table.

Using the Violations Table

The following rules concern the structure and use of the violations table:

- Every pair of update rows in the violations table has the same value in the **informix_tupleid** column to indicate that both rows refer to the same row in the target table.
- If the target table has columns named **informix_tupleid**, **informix_optype**, or **informix_recowner**, the database server attempts to generate alternative names for these columns in the violations table by appending a digit to the end of the column name (for example, **informix_tupleid1**). If this attempt fails, the database server returns an error, and the violations table is not started for the target table.
- When a table functions as a violations table, it cannot have triggers or constraints defined on it.
- When a table functions as a violations table, users can create indexes on the table, even though the existence of an index affects performance. Unique indexes on the violations table cannot be set to the filtering object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in cascade mode (the default mode) causes the violations and diagnostics tables to be dropped also. If the target table is dropped in the restricted mode, the existence of the violations and diagnostics tables causes the DROP TABLE statement to fail.
- Once a violations table is started for a target table, you cannot use the ALTER TABLE statement to add, modify, or drop columns in the target table, violations table, or diagnostics table. Before you can alter any of these tables, you must issue a STOP TABLE VIOLATIONS statement for the target table.
- The database server does not clear out the contents of the violations table before or after it uses the violations table during an Insert, Update, Delete, or Set operation.

- If a target table has a filtering-mode constraint or unique index defined on it and a violations table associated with it, users cannot insert into the target table by selecting from the violations table. Before you insert rows into the target table by selecting from the violations table, you must take one of the following steps:
  - You can set the object mode of the constraint or unique index to the enabled or disabled object mode.
  - You can issue a STOP VIOLATIONS TABLE statement for the target table.

  If it is inconvenient to take either of these steps, but you still want to copy records from the violations table into the target table, a third option is to select from the violations table into a temporary table and then insert the contents of the temporary table into the target table.
- If the target table that is specified in the START VIOLATIONS TABLE statement is fragmented, the violations table has the same fragmentation strategy as the target table. Each fragment of the violations table is stored in the same dbspace as the corresponding fragment of the target table.
- If the target table specified in the START VIOLATIONS TABLE statement is not fragmented, the database server places the violations table in the same dbspace as the target table.
- If the target table has blob columns, blobs in the violations table are created in the same blob space as the blobs in the target table.

Example of a Violations Table

To start a violations and diagnostics table for the target table named **customer** in the **stores7** demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR customer
```

Because your START VIOLATIONS statement does not include a USING clause, the violations table is named **customer_vio** by default. The **customer_vio** table includes the following columns:

```
customer_num
fname
lname
company
address1
address2
city
state
zipcode
phone
informix_tupleid
informix_optype
informix_recowner
```

The **customer_vio** table has the same table definition as the **customer** table except that the **customer_vio** table has three additional columns that contain information about the operation that caused the bad row.

Structure of the Diagnostics Table

When you issue a START VIOLATIONS TABLE statement for a target table, the diagnostics table that the statement creates has a predefined structure. This structure is independent of the structure of the target table.

The following table shows the structure of the diagnostics table.

Column Name	Type	Explanation
informix_tupleid	INTEGER	This column in the diagnostics table implicitly refers to the values in the **informix_tupleid** column in the violations table. However, this relationship is not declared as a foreign-key to primary-key relationship.
objtype	CHAR(1)	This column identifies the type of the violation. This column can have the following values. C = Constraint violation I = Unique-index violation

(1 of 2)

Column Name	Type	Explanation
objowner	CHAR(8)	This column identifies the owner of the constraint or index for which an integrity violation was detected.
objname	CHAR(18)	This column contains the name of the constraint or index for which an integrity violation was detected.

(2 of 2)

Initial Privileges on the Diagnostics Table

When the START VIOLATIONS TABLE statement creates the diagnostics table, the database server uses the set of privileges granted on the target table as a basis for granting privileges on the diagnostics table. However, the database server follows different rules when it grants each type of privilege.

The following table shows the initial set of privileges on the diagnostics table. The **Privilege** column lists the privilege. The **Condition** column explains the conditions under which the database server grants the privilege to a user.

Privilege	Condition
Insert	The user has the Insert privilege on the diagnostics table if the user has any of the following privileges on the target table: the Insert privilege, the Delete privilege, or the Update privilege on any column.
Delete	The user has the Delete privilege on the diagnostics table if the user has any of the following privileges on the target table: the Insert privilege, the Delete privilege, or the Update privilege on any column.
Select	The user has the Select privilege on the diagnostics table if the user has the Select privilege on any column in the target table.
Update	The user has the Update privilege on the diagnostics table if the user has the Update privilege on any column in the target table.

(1 of 2)

Privilege	Condition
Index	The user has the Index privilege on the diagnostics table if the user has the Index privilege on the target table.
Alter	The Alter privilege is not granted on the diagnostics table. (Users cannot alter diagnostics tables.)
References	The References privilege is not granted on the diagnostics table. (Users cannot add referential constraints to diagnostics tables.)

(2 of 2)

The following rules concern privileges on the diagnostics table:

- When the diagnostics table is created, the owner of the target table becomes the owner of the diagnostics table.
- The owner of the diagnostics table automatically receives all table-level privileges on the diagnostics table, including the Alter and References privileges. However, the database server prevents the owner of the diagnostics table from altering the diagnostics table or adding a referential constraint to the diagnostics table.
- You can use the GRANT and REVOKE statements to modify the initial set of privileges on the diagnostics table.
- When you issue an INSERT, DELETE, or UPDATE statement on a target table that has a filtering-mode unique index or constraint defined on it, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the INSERT, DELETE, or UPDATE statement on the target table provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the INSERT, DELETE, or UPDATE statement.

  Similarly, when you issue a SET statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET statement provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of insert permission on the violations and diagnostics tables unless an integrity violation is detected during the execution of the SET statement.

- The grantor of the initial set of privileges on the diagnostics table is the same as the grantor of the privileges on the target table. For example, if the user **jenny** has been granted the Insert privilege on the target table by both the user **wayne** and the user **laurie**, both user **wayne** and user **laurie** grant the Insert privilege on the diagnostics table to user **jenny**.
- Once a diagnostics table has been started for a target table, revoking a privilege on the target table from a user does not automatically revoke the same privilege on the diagnostics table from that user. Instead you must explicitly revoke the privilege on the diagnostics table from the user.
- If you have fragment-level privileges on the target table, you have the corresponding table-level privileges on the diagnostics table.

Example of Privileges on the Diagnostics Table

The following example illustrates how the initial set of privileges on a diagnostics table is derived from the current set of privileges on the target table.

For example, assume that there is a table called **cust_subset** and that this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives).

The following set of privileges exists on the **cust_subset** table:

- User **alvin** is the owner of the table.
- User **barbara** has the Insert and Index privileges on the table. She also has the Select privilege on the **ssn** and **lname** columns.

- User **carrie** has the Update privilege on the **city** column. She also has the Select privilege on the **ssn** column.
- User **danny** has the Alter privilege on the table.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table, as follows:

```
START VIOLATIONS TABLE FOR cust_subset
    USING cust_subset_viols, cust_subset_diags
```

The database server grants the following set of initial privileges on the **cust_subset_diags** diagnostics table:

- User **alvin** is the owner of the diagnostics table, so he has all table-level privileges on the table.
- User **barbara** has the Insert, Delete, Select, and Index privileges on the diagnostics table.
- User **carrie** has the Insert, Delete, Select, and Update privileges on the diagnostics table.
- User **danny** has no privileges on the diagnostics table.

Using the Diagnostics Table

For information on the relationship between the diagnostics table and the violations table, see "Relationship Between the Violations and Diagnostics Tables" on page 1-587.

The following issues concern the structure and use of the diagnostics table:

- The MAX ROWS clause of the START VIOLATIONS TABLE statement sets a limit on the number of rows that can be inserted into the diagnostics table when you execute a single statement, such as an INSERT or SET statement, on the target table.
- The MAX ROWS clause limits the number of rows only for operations in which the table functions as a diagnostics table.
- When a table functions as a diagnostics table, it cannot have triggers or constraints defined on it.

- When a table functions as a diagnostics table, users can create indexes on the table, even though the existence of an index affects performance. You cannot set unique indexes on the diagnostics table to the filtering object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in the cascade mode (the default mode) causes the violations and diagnostics tables to be dropped also. If the target table is dropped in the restricted mode, the existence of the violations and diagnostics tables causes the DROP TABLE statement to fail.
- Once a violations table is started for a target table, you cannot use the ALTER TABLE statement to add, modify, or drop columns in the target table, violations table, or diagnostics table. Before you can alter any of these tables, you must issue a STOP TABLE VIOLATIONS statement for the target table.
- The database server does not clear out the contents of the diagnostics table before or after it uses the diagnostics table during an Insert, Update, Delete, or Set operation.
- If the target table that is specified in the START VIOLATIONS TABLE statement is fragmented, the diagnostics table is fragmented with a round-robin strategy over the same dbspaces in which the target table is fragmented.

Example of a Diagnostics Table

To start a violations and diagnostics table for the target table named **stock** in the **stores7** demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR stock
```

Because your START VIOLATIONS TABLE statement does not include a USING clause, the diagnostics table is named **stock_dia** by default. The **stock_dia** table includes the following columns:

```
informix_tupleid
objtype
objowner
objname
```

This list of columns shows an important difference between the diagnostics table and violations table for a target table. Whereas the violations table has a matching column for every column in the target table, the columns of the diagnostics table do not match any columns in the target table. The diagnostics table created by any START VIOLATIONS TABLE statement always has the same columns with the same column names and data types.

References

See the STOP VIOLATIONS TABLE and SET statements in this manual.

For information on the system catalog tables that are associated with the START VIOLATIONS TABLE statement, see the **sysobjstate** and **sysviolations** tables in the *Informix Guide to SQL: Reference*.

STOP VIOLATIONS TABLE

The STOP VIOLATIONS TABLE statement drops the association between a target table and the special violations and diagnostics tables.

Syntax

Element	Purpose	Restrictions	Syntax
table name	The name of the target table whose association with the violations and diagnostics table is to be dropped. There is no default value.	The target table must have a violations and diagnostics table associated with it before you can execute the statement. The target table must be a local table.	Identifier, p. 1-723

Usage

The STOP VIOLATIONS TABLE statement drops the association between the target table and the violations and diagnostics tables. After you issue this statement, the former violations and diagnostics tables continue to exist, but they no longer function as violations and diagnostics tables for the target table. They now have the status of regular database tables instead of violations and diagnostics tables for the target table. You must issue the DROP TABLE statement to drop these two tables explicitly.

When Insert, Delete, and Update operations cause data-integrity violations for rows of the target table, the nonconforming rows are no longer filtered to the former violations table, and diagnostics information about the data-integrity violations is not placed in the former diagnostics table.

Example of Stopping a Violations and Diagnostics Table

Assume that a target table named **cust_subset** has an associated violations table named **cust_subset_vio** and an associated diagnostics table named **cust_subset_dia**. To drop the association between the target table and the violations and diagnostics tables, enter the following statement:

```
STOP VIOLATIONS TABLE FOR cust_subset
```

Example of Dropping a Violations and Diagnostics Table

After you execute the STOP VIOLATIONS TABLE statement in the preceding example, the **cust_subset_vio** and **cust_subset_dia** tables continue to exist, but they are no longer associated with the **cust_subset** table. Instead they now have the status of regular database tables. To drop these two tables, enter the following statements:

```
DROP TABLE cust_subset_vio;
DROP TABLE cust_subset_dia;
```

Privileges Required for Stopping a Violations Table

To stop a violations and diagnostics table for a target table, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the target table and have the Resource privilege on the database.
- You must have the Alter privilege on the target table and the Resource privilege on the database.

References

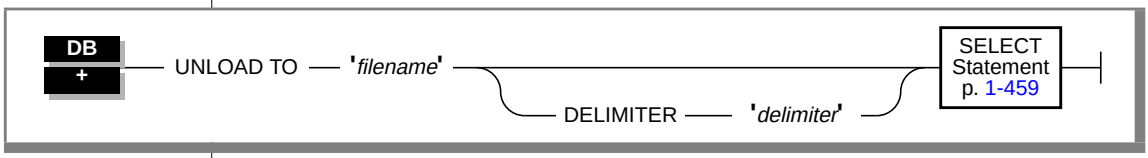
See the SET and START VIOLATIONS TABLE statements in this manual.

For information on the system catalog tables associated with the STOP VIOLATIONS TABLE statement, see the **sysobjstate** and **sysviolations** tables in the *Informix Guide to SQL: Reference*.

UNLOAD

Use the UNLOAD statement to write the rows retrieved in a SELECT statement to an operating-system file.

Syntax



Element	Purpose	Restrictions	Syntax
delimiter	A quoted string that identifies the character to use as the delimiter in the output file. The delimiter is a character that separates the data values in each line of the output file. If you do not specify a delimiter character, the database server uses the setting in the **DBDELIMITER** environment variable. If **DBDELIMITER** has not been set, the default delimiter is the vertical bar (|).	You cannot use the following items as the delimiter character: backslash (\ ), new-line character (=CTRL-J), hexadecimal numbers (0 to 9, a to f, A to F).	Quoted String, p. 1-757
filename	A quoted string that specifies the pathname and filename of an ASCII operating-system file. This output file receives the selected rows from the table during the unload operation. The default pathname for the output file is the current directory.	You can unload table data containing VARCHAR or BLOB data types to the output file, but you should be aware of the consequences. See "The UNLOAD TO File" on page 1-605 for further information.	Quoted String, p. 1-757. The pathname and filename specified in the quoted string must conform to the conventions of your operating system.

Usage

To use the UNLOAD statement, you must have the Select privilege on all columns selected in the SELECT statement. For information on database-level and table-level privileges, see the GRANT statement on page 1-340.

The SELECT statement can consist of a literal SELECT statement or the name of a character variable that contains a SELECT statement. (See the SELECT statement on page 1-459.)

The UNLOAD TO File

The UNLOAD TO file contains the selected rows retrieved from the table. You can use the UNLOAD TO file as the LOAD FROM file in a LOAD statement.

The following table shows types of data and their output format for an UNLOAD statement in DB-Access (when DB-Access uses the default locale, U.S. English).

Data Type	Output Format
character	If a character field contains the delimiter character, Informix products automatically escape it with a backslash (\) to prevent interpretation as a special character. (If you use a LOAD statement to insert the rows into a table, backslashes are automatically stripped.) Trailing blanks are automatically clipped.
date	DATE values are represented as *mm* / *dd* / *yyyy*, where *mm* is the month (January = 1, and so on), *dd* is the day, and *yyyy* is the year. If you have set the **GL_DATE** or **DBDATE** environment variable, the UNLOAD statement uses the specified date format for DATE values. See Chapter 2 of the *Guide to GLS Functionality* for more information about these environment variables.
MONEY	MONEY values are unloaded with no leading currency symbol. They use the comma (,) as the thousands separator and the period as the decimal separator. If you have set the **DBMONEY** environment variable, the UNLOAD statement uses the specified currency format for MONEY values. See Chapter 2 of the *Guide to GLS Functionality* for more information about this environment variable.

(1 of 2)

Data Type	Output Format
NULL	NULL columns are unloaded by placing no characters between the delimiters.
number	Number data types are displayed with no leading blanks. INTEGER or SMALLINT zero are represented as 0, and FLOAT, SMALLFLOAT, DECIMAL, or MONEY zero are represented as 0.00.
time	DATETIME and INTERVAL values are represented in character form, showing only their field digits and delimiters. No type specification or qualifiers are included in the output. The following pattern is used: *yyyy-mm-dd hh:mi:ss.fff*, omitting fields that are not part of the data. If you have set the **GL_DATETIME** or **DBTIME** environment variable, the UNLOAD statement uses the specified format for DATETIME values. See Chapter 2 of the *Guide to GLS Functionality* for more information on these environment variables.

(2 of 2)

GLS

If you are using a nondefault locale, the formats of DATE, DATETIME, MONEY, and numeric column values in the UNLOAD TO file are determined by the formats that the locale supports for these data types. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

Do not use the backslash (\) as a field separator or UNLOAD delimiter. It serves as an escape character to inform the UNLOAD command that the next character is to be interpreted as part of the data.

If you are unloading files containing VARCHAR or BLOB data types, note the following information:

- BYTE items are written in hexadecimal dump format with no added spaces or new lines. Consequently, the logical length of an unloaded file that contains BYTE items can be very long and very difficult to print or edit.
- Trailing blanks are retained in VARCHAR fields.
- Do not use the following characters as delimiters in the UNLOAD TO file: `0` to `9`, `a` to `f`, `A` to `F`, new-line character, or backslash.

If you are unloading files that contain BLOB data types, blobs smaller than 10 kilobytes are stored temporarily in memory. You can adjust the 10-kilobyte setting to a larger setting with the **DBBLOBBUF** environment variable. Blobs larger than the default or the setting of the **DBBLOBBUF** environment variable are stored in a temporary file. For additional information about the **DBBLOBBUF** environment variable, see the *Informix Guide to SQL: Reference*.

The following statement unloads rows from the **customer** table where the value of **customer_num** is greater than or equal to 138, and puts them in a file named **cust_file**:

```
UNLOAD TO 'cust_file' DELIMITER '!'
    SELECT * FROM customer WHERE customer_num> = 138
```

The output file, **cust_file**, appears as shown in the following example:

```
138!Jeffery!Padgett!Wheel Thrills!3450 El Camino!Suite
10!Palo Alto!CA!94306!!
139!Linda!Lane!Palo Alto Bicycles!2344 University!!Palo
Alto!CA!94301!(415)323-5400
```

DELIMITER Clause

Use the DELIMITER clause to identify the delimiter that separates the data contained in each column in a row in the output file. If you omit this clause, DB-Access checks the **DBDELIMITER** environment variable.

If **DBDELIMITER** has not been set, the default delimiter is the vertical bar (|). See Chapter 4 of the *Informix Guide to SQL: Reference* for information about setting the **DBDELIMITER** environment variable.

You can specify the TAB (= CTRL-I) or <blank> (= ASCII 32) as the delimiter symbol. You cannot use the following as the delimiter symbol:

- Backslash (\)
- New-line character (= CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

The following statement specifies the semicolon (;) as the delimiter character:

```
UNLOAD TO 'cust.out' DELIMITER ';'
    SELECT fname, lname, company, city
        FROM customer
```

References

See the LOAD and SELECT statements in this manual.

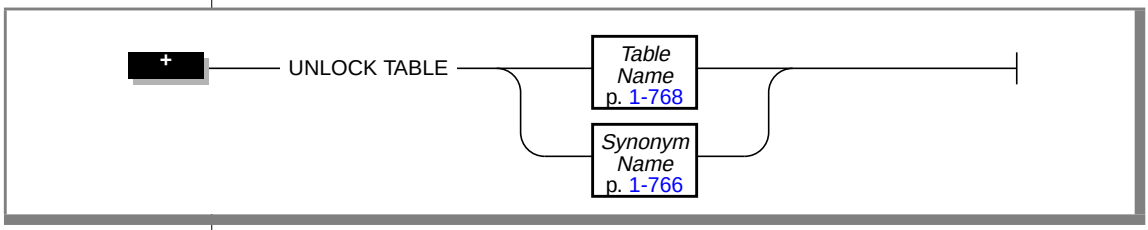
In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the UNLOAD statement

In the *Informix Migration Guide*, see the task-oriented discussion of the UNLOAD statement and other utilities for moving data.

UNLOCK TABLE

Use the UNLOCK TABLE statement in a database without transactions to unlock a table that you previously locked with the LOCK TABLE statement.

Syntax



Usage

You can lock a table if you own the table or if you have the Select privileges on the table, either from a direct grant or from a grant to **public**. You can only unlock a table that you locked. You cannot unlock a table that another process locked. Only one lock can apply to a table at a time.

The *table name* either is the name of the table you are unlocking or a synonym for the table. Do not specify a view or a synonym of a view.

To change the lock mode of a table in a database without transactions, use the UNLOCK TABLE statement to unlock the table, then issue a new LOCK TABLE statement.

The UNLOCK TABLE statement fails if it is issued within a transaction. Table locks set within a transaction are released automatically when the transaction completes.

ANSI

You should not issue an UNLOCK TABLE statement within an ANSI-compliant database. The UNLOCK TABLE statement fails if it is issued within a transaction, and a transaction is always in effect in an ANSI-compliant database. ♦

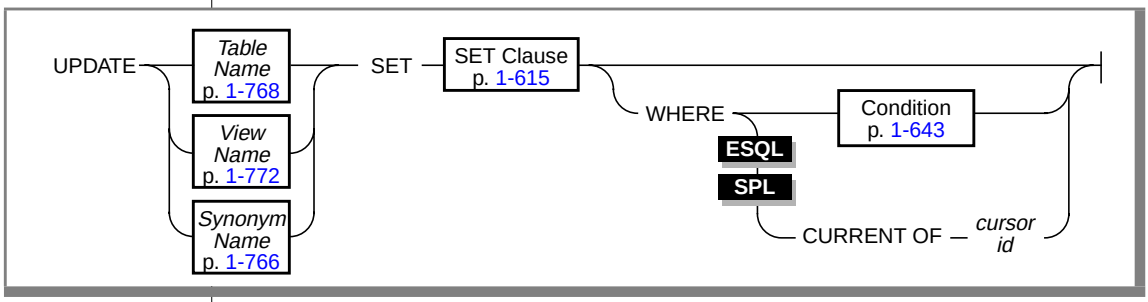
References

See the COMMIT WORK, ROLLBACK WORK, and LOCK TABLE statements in this manual.

UPDATE

Use the UPDATE statement to change the values in one or more columns of one or more rows in a table or view.

Syntax



Element	Purpose	Restrictions	Syntax
cursor id	The name of the cursor to be used by the UPDATE statement. The current row of the active set for this cursor is updated when the UPDATE statement is executed. See "WHERE CURRENT OF Clause" on page 1-619 for more information on this parameter.	You cannot update a row with a cursor if that row includes aggregates. The specified cursor (as defined in the SELECT...FOR UPDATE portion of a DECLARE statement) can contain only column names. If the cursor was created without specifying particular columns for updating, you can update any column in a subsequent UPDATE...WHERE CURRENT OF statement. But if the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause, you can update only those columns in a subsequent UPDATE...WHERE CURRENT OF statement.	Identifier, p. 1-723

Usage

To update data in a table, you must either own the table or have the Update privilege for the table (see the GRANT statement on page 1-340). To update data in a view, you must have the Update privilege, and the view must meet the requirements that are explained in "Updating Rows Through a View" below.

If you omit the WHERE clause, all rows of the target table are updated.

If you are using effective checking, and the checking mode is set to IMMEDIATE, all specified constraints are checked at the end of each UPDATE statement. If the checking mode is set to DEFERRED, all specified constraints are *not* checked until the transaction is committed.

DB

If you omit the WHERE clause and are in interactive mode, DB-Access does not run the UPDATE statement until you confirm that you want to change all rows. However, if the statement is in a command file, and you are running from the command line, the statement executes immediately. ♦

Updating Rows Through a View

You can update data through a *single-table* view if you have the Update privilege on the view (see the GRANT statement on page 1-340). To do this, the defining SELECT statement can select from only one table, and it cannot contain any of the following elements:

- DISTINCT keyword
- GROUP BY clause
- Derived value (also called a virtual column)
- Aggregate value

You can use data-integrity constraints to prevent users from updating values in the underlying table when the update values do not fit the SELECT statement that defined the view. For further information, refer to the WITH CHECK OPTION discussion in the CREATE VIEW statement on page 1-224.

Because duplicate rows can occur in a view even though the underlying table has unique rows, be careful when you update a table through a view. For example, if a view is defined on the **items** table and contains only the **order_num** and **total_price** columns, and if two items from the same order have the same total price, the view contains duplicate rows. In this case, if you update one of the two duplicate total price values, you have no way to know which item price is updated.

Important: *You cannot update rows to a remote table through views with check options.*

Updating Rows in a Database Without Transactions

If you are updating rows in a database without transactions, you must take explicit action to restore updated rows. For example, if the UPDATE statement fails after updating some rows, the successfully updated rows remain in the table. You cannot automatically recover from a failed update.

Updating Rows in a Database with Transactions

If you are updating rows in a database with transactions, and you are using transactions, you can undo the update using the ROLLBACK WORK statement. If you do not execute a BEGIN WORK statement before the update, and the update fails, the database server automatically rolls back any database modifications made since the beginning of the update.

ANSI

If you are updating rows in an ANSI-compliant database, transactions are implicit, and all database modifications take place within a transaction. In this case, if an UPDATE statement fails, you can use the ROLLBACK WORK statement to undo the update.

If you are using INFORMIX-OnLine Dynamic Server, you are within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update. ♦

Locking Considerations

If you are using an OnLine database server, when a row is selected with the intent to update, the update process acquires an update lock. Update locks permit other processes to read, or *share*, a row that is about to be updated but do not let those processes update or delete it. Just before the update occurs, the update process *promotes* the shared lock to an exclusive lock. An exclusive lock prevents other processes from reading or modifying the contents of the row until the lock is released.

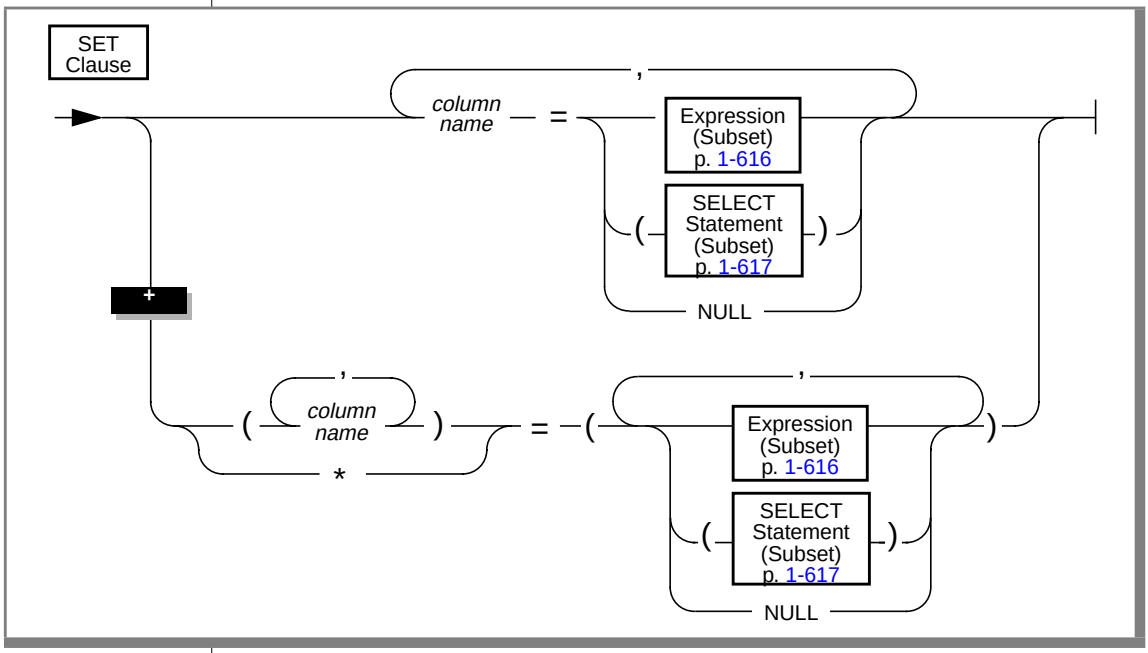
INFORMIX-OnLine Dynamic Server allows only one update lock at a time on a row or a page (the type of lock depends on the lock mode that is selected in the CREATE TABLE or ALTER TABLE statements). An update process can acquire an update lock on a row or a page that has a shared lock from another process, but you cannot promote the update lock from shared to exclusive (and the update cannot occur) until the other process releases its lock.

If the number of rows affected by a single update is very large, you can exceed the limits placed on the maximum number of simultaneous locks. If this occurs, you can reduce the number of transactions per UPDATE statement, or you can lock the page (OnLine database servers only) or the entire table before you execute the statement.

SE

Individual rows of a table are locked automatically when you execute an UPDATE statement. ♦

SET Clause


SET Clause
column name
=
Expression (Subset) p. 1-616
SELECT Statement (Subset) p. 1-617
NULL
+
column name
*
=
Expression (Subset) p. 1-616
SELECT Statement (Subset) p. 1-617
NULL

Element	Purpose	Restrictions	Syntax
*			

The SET clause identifies the columns to be updated and assigns values to each column. The clause either pairs a single column to a single expression or lists multiple columns and sets them equal to corresponding expressions.

Subset of Expressions Allowed in the SET Clause

You cannot use an expression comprised of aggregate functions in the SET clause. For a complete description of syntax and usage, see the Expression segment on page 1-671.

Subset of SELECT Statements Allowed in the SET Clause

A SELECT statement used in a SET clause can return more than *one column* of information in a row. However, the SELECT statement cannot return more than *one row* of information in a table. For a complete description of syntax and usage, refer to the SELECT statement on page 1-459.

Single Columns to Single Expressions

You can include any number of single-column to single-expressions in an UPDATE statement.

The following examples illustrate the single-column to single-expression form of the SET clause:

```
UPDATE customer
    SET address1 = '1111 Alder Court',
        city = 'Palo Alto',
        zipcode = '94301'
    WHERE customer_num = 103

UPDATE orders
    SET ship_charge =
        (SELECT SUM(total_price) * .07
            FROM items
            WHERE orders.order_num = items.order_num)
        WHERE orders.order_num = 1001

UPDATE stock
    SET unit_price = unit_price * 1.07
```

Updating a Column to NULL

You can use the NULL keyword to modify a column value when you use the UPDATE statement. For a customer whose previous address required two address lines but now requires only one, you would use the following entry:

```
UPDATE customer
    SET address1 = '123 New Street',
    SET address2 = null,
    city = 'Palo Alto',
    zipcode = '94303'
    WHERE customer_num = 134
```

Multiple Columns Equal to Multiple Expressions

The SET clause offers the following options for listing a series of columns you intend to update:

- Explicitly list each column, placing commas between columns and enclosing the set of columns in parentheses.
- Implicitly list all columns in *table name* using the asterisk notation (*).

To complete the SET clause, you must list each expression explicitly, placing commas between expressions and enclosing the set of expressions in parentheses. An expression list can include an <vk>SQL subquery that returns a single row of multiple values as long as the number of columns named, explicitly or implicitly, equals the number of values produced by the expression or expressions that follow the equal sign.

The following examples illustrate the multiple-column to multiple-expression form of the SET clause:

```
UPDATE customer
    SET (fname, lname) = ('John', 'Doe')
    WHERE customer_num = 101


UPDATE manufact
    SET * = ('HNT', 'Hunter')
    WHERE manu_code = 'ANZ'


UPDATE items
    SET (stock_num, manu_code, quantity) =
        ( (SELECT stock_num, manu_code FROM stock
            WHERE description = 'baseball'), 2)
    WHERE item_num = 1 AND order_num = 1001


UPDATE table1
    SET (col1, col2, col3) =
        ((SELECT MIN (ship_charge),
            MAX (ship_charge) FROM orders),
            '07/01/1993')
    WHERE col4 = 1001
```

WHERE Clause

The WHERE clause lets you limit the rows that you want to update. If you omit the WHERE clause, every row in the table is updated.

The WHERE clause consists of a standard search condition. (For more information, see the SELECT statement on page 1-459). The following example illustrates a WHERE condition within an UPDATE statement. In this example, the statement updates three columns (**state**, **zipcode**, and **phone**) in each row of the **customer** table that has a corresponding entry in a table of new addresses called **new_address**.

```
UPDATE customer
    SET (state, zipcode, phone) =
        ((SELECT state, zipcode, phone FROM new_address N
            WHERE N.cust_num =
                  customer.customer_num))
        WHERE customer_num IN
            (SELECT cust_num FROM new_address)
```

When you use the UPDATE statement with the WHERE clause, and no rows are updated, the SQLNOTFOUND value is 100 in ANSI-compliant databases and 0 in databases that are not ANSI compliant. If the UPDATE ... WHERE ... is a part of a multistatement prepare, and no rows are returned, the SQLNOTFOUND value is 100 for ANSI-compliant databases and databases that are not ANSI compliant.

WHERE CURRENT OF Clause

ESQL

You can use the CURRENT OF keyword to update the current row of the active set of a cursor. However, you cannot update a row with a cursor if that row includes aggregates. The cursor named in the CURRENT OF clause can only contain column names. The UPDATE statement does not advance the cursor to the next row, so the current row position remains unchanged.

You can restrict the effect of the CURRENT OF keyword if you associate the UPDATE statement with a cursor that was created with the FOR UPDATE keyword. (See the DECLARE statement on page 1-234.) If you created the cursor without specifying any columns for updating, you can update any column in a subsequent UPDATE...WHERE CURRENT OF statement. However, if the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause, you are restricted to updating only those columns in a subsequent UPDATE...WHERE CURRENT OF statement. The advantage to specifying columns in the FOR UPDATE clause of a DECLARE statement is speed. INFORMIX-SE and INFORMIX-OnLine Dynamic Server can usually perform updates more quickly if columns are specified in the DECLARE statement. ♦

The following INFORMIX-ESQL/C example illustrates the WHERE CURRENT OF form of the WHERE clause. In this example, updates are performed on a range of customers who receive 10-percent discounts (assume that a new column, **discount**, is added to the **customer** table). The UPDATE statement is prepared outside the WHILE loop to ensure that parsing is done only once. (For more information, see the PREPARE statement on page 1-402.)

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
    char fname[32],lname[32];
    int low,high;
EXEC SQL END DECLARE SECTION;

main()
{
    EXEC SQL connect to 'stores7';
    EXEC SQL prepare sel_stmt from
        'select fname, lname from customer \
         where cust_num between ? and ? for update';
    EXEC SQL declare x cursor for sel_stmt;
    printf("\nEnter lower limit customer number: ");
    scanf("%d", &low);
    printf("\nEnter upper limit customer number: ");
    scanf("%d", &high);
    EXEC SQL open x using :low, :high;
    EXEC SQL prepare u from
        'update customer set discount = 0.1 \
        where current of x';

    while (1)
        {
        EXEC SQL fetch x into :fname, :lname;
        if ( SQLCODE == SQLNOTFOUND)
            break;
```

```
            }
        printf("\nUpdate %.10s %.10s (y/n)?", fname, lname);
        if (answer = getch() == 'y')
            EXEC SQL execute u;
        EXEC SQL close x;
    }
```

Tip: *You can use an update cursor to perform updates that are not possible with the UPDATE statement. An update cursor is a sequential cursor that is associated with a SELECT statement, which is declared with the FOR UPDATE keyword. For more information on the update cursor, see page 1-239.*

References

See the DECLARE, INSERT, OPEN, and SELECT statements in this manual.

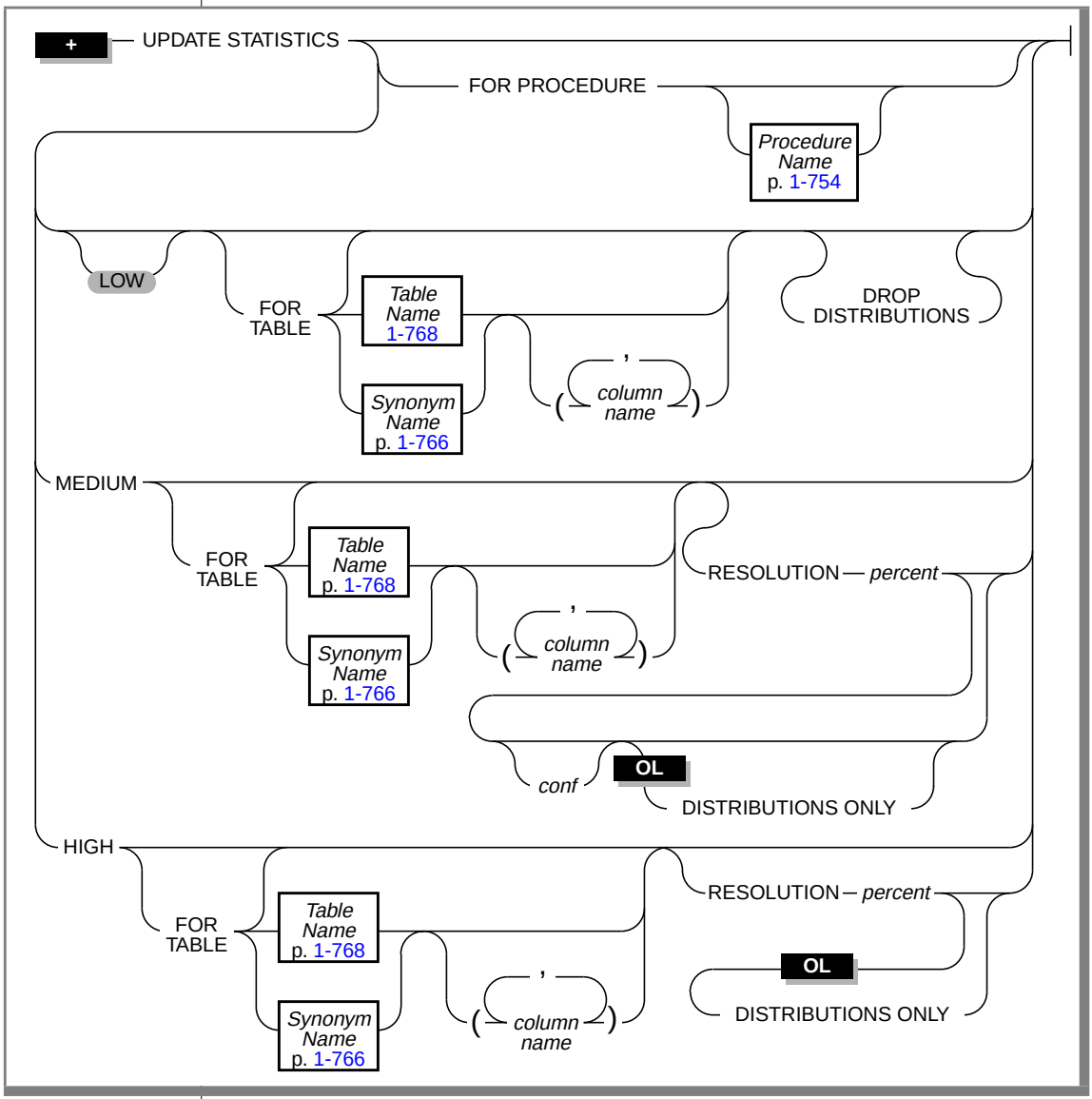
In the *Informix Guide to SQL: Tutorial*, see the discussion of the UPDATE statement in Chapter 6.

In the *Guide to GLS Functionality*, see the discussion of the GLS aspects of the UPDATE statement.

UPDATE STATISTICS

Use the UPDATE STATISTICS statement to update system catalog tables with information used to determine optimal query plans. In addition, you can use the UPDATE STATISTICS statement to force stored procedures to be reoptimized. If you are upgrading to a new version of the database server, you can use UPDATE STATISTICS to convert table indexes to the format that the new database server uses.

Syntax


+
UPDATE STATISTICS
FOR PROCEDURE
Procedure Name
p. 1-754
LOW
FOR TABLE
Table Name
1-768
Synonym Name
p. 1-766
,
column name
DROP DISTRIBUTIONS
MEDIUM
FOR TABLE
Table Name
p. 1-768
Synonym Name
p. 1-766
,
column name
RESOLUTION
percent
conf
OL
DISTRIBUTIONS ONLY
HIGH
FOR TABLE
Table Name
p. 1-768
Synonym Name
p. 1-766
,
column name
RESOLUTION
percent
OL
DISTRIBUTIONS ONLY

Element	Purpose	Restrictions	Syntax
column name	The name of a column in the specified table	The column must exist. If you use the LOW keyword and want the UPDATE STATISTICS statement to do minimal work, specify a column name that is not part of an index. If you use the MEDIUM or HIGH keywords, *column name* cannot be a BYTE or TEXT column.	Identifier, p. 1-723
conf	The expected fraction of times that the sampling entailed by the MEDIUM keyword should produce the same results as the exact methods entailed by the HIGH keyword. The default confidence level is 0.95. See "Specifying MEDIUM Distributions" on page 1-628 for further information on this parameter.	The minimum confidence level is 0.80. The maximum confidence level is 0.99.	Literal Number, p. 1-752
percent	The desired resolution in units of percent, so that 0.1 means the data in a column is divided into bins, each containing (on average) 0.1 percent of the data. The default value of *percent* is 2.5 when the MEDIUM keyword is used. The default value of *percent* is 0.5 when the HIGH keyword is used. See "Creating Distributions for Columns" on page 1-627 for further information on this parameter.	The minimum resolution possible for a table is 1 / *nrows*, where *nrows* is the number of rows in the table.	Literal Number, p. 1-752

Usage

When you issue an UPDATE STATISTICS statement, INFORMIX-OnLine Dynamic Server recalculates the data in the **systables**, **syscolumns**, **sysindexes**, and **sysdistrib** system catalog tables. The optimizer uses this data to determine the best execution path for queries. The database server does not update this statistical data automatically. Statistics are updated only when you issue an UPDATE STATISTICS statement.

Using the UPDATE STATISTICS statement also updates the optimized execution plans for procedures in the **sysprocplan** system catalog table. Each time a procedure executes, the database server reoptimizes its execution plan if any objects that are referenced in the procedure have changed.

The UPDATE STATISTICS statement requires a current database. If you omit the FOR TABLE or FOR PROCEDURE clauses, statistics are updated for every table and procedure in the current database, including the system tables.

If you use UPDATE STATISTICS ... FOR TABLE without a table name, the statistics for all tables, including temporary tables, in the current database are updated. If you use the FOR PROCEDURE keyword without a procedure name, the statistics for all stored procedures in the current database are updated.

You cannot update the statistics used by the optimizer for a table or procedure that is external to the current database.

SE

With INFORMIX-SE, UPDATE STATISTICS does not update rows in the **sysindexes** table; when you issue an UPDATE STATISTICS statement, INFORMIX-SE recalculates only the data in the **systables** and, when requested, the **sysdistrib** system catalog tables. ♦

Examining Index Pages

The UPDATE STATISTICS statement reads through index pages in order to compute statistics for the query optimizer. In addition, the statement also looks for pages that have the delete flag marked as one. If pages are found with the delete flag marked as one, the keys so marked are removed from the btree cleaner list. This is particularly useful if a system crash causes the btree cleaner list to be lost (because it is in shared memory). To remove those items, run the UPDATE STATISTICS statement. For information on the btree cleaner list, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

When to Update Statistics

Update the statistics when you perform extensive modifications to a table or when changes are made to tables that are used by one or more procedures, and you do not want the database server to reoptimize the procedure at execution time.

If your application makes many modifications to the data in a particular table, update the system catalog table data for that table routinely with the UPDATE STATISTICS statement to improve the efficiency of queries. *Many* is relative to the resolution of the distributions. In addition, if the data changes do not change the distribution of column values, you do not need to execute UPDATE STATISTICS again.

Using UPDATE STATISTICS when Upgrading the Database Server

When you are upgrading a database for use with a newer database server, you can use the UPDATE STATISTICS statement to convert the indexes to the form that the newer server uses. You can choose to convert the indexes table by table or for the entire database at one time. You should follow the conversion guidelines that are outlined in the *Informix Migration Guide* or the *INFORMIX-SE Administrator's Guide*.When you use the UPDATE STATISTICS statement to convert the indexes for use with a newer database server, the indexes are implicitly dropped and re-created. Upgrading is the only time that an UPDATE STATISTICS statement causes implicit dropping and recreation of table indexes.

Specifying the LOW Keyword

If you use the LOW keyword, or if you specify no keyword, the smallest amount of information is gathered about the column. The data in the **systables**, **syscolumns**, and **sysindexes** tables is updated. No information is put into the **sysdistrib** system catalog table. If data already exists in the **sysdistrib** system catalog table when you run an UPDATE STATISTICS (LOW) statement, the distribution data remains intact unless the DROP DISTRIBUTIONS option is used. If the DROP DISTRIBUTIONS option is specified, but no table name is specified, all the distribution information is removed.

The UPDATE STATISTICS (LOW) statement updates table, row, and page counts as well as index and column statistics for specified columns. If you want the UPDATE STATISTICS statement to do minimal work, specify a column that is not part of an index.

The following example updates statistics on the **customer_num** column of the **customer** table. All distributions associated with the **customer** table remain intact, even those that already exist on the **customer_num** column.

```
UPDATE STATISTICS LOW FOR TABLE customer (customer_num)
```

Dropping Data with the DROP DISTRIBUTIONS Clause

If you want to drop distribution data for some or all the columns that are already defined in the **sysdistrib** table, but you want to update the statistics with the LOW option for the rest of the columns in the table, you can use the DROP DISTRIBUTIONS clause. If you specify the DROP DISTRIBUTIONS keyword, all distribution information that exists for the column specified in the UPDATE STATISTICS statement drops. If no columns are specified, all the distributions for that table are removed.

You must have the DBA privileges or be the owner of the table in order to drop distributions.

The following example shows how to remove distributions for the **customer_num** column in the **customer** table:

```
UPDATE STATISTICS LOW
FOR TABLE customer (customer_num) DROP DISTRIBUTIONS
```

Creating Distributions for Columns

Distributions are a mapping of the data in the column into a carefully chosen set of the column values. The contents of the column are examined and divided into bins, which represent a percentage of data. For example, a bin might hold 2 percent of the data; 50 bins would hold all the data. You can set the width of the bin with the RESOLUTION *percent* parameter.

This organization of column values into bins is called the distribution (for that column). The optimizer examines distributions of columns that are referenced in a WHERE clause to estimate the effect of a WHERE clause on the data.

You cannot create distributions for TEXT or BYTE columns. If you include a TEXT or BYTE column in an UPDATE STATISTICS statement that specifies medium or high distributions, no distributions are created for those columns. Distributions are constructed for other columns in the list, and the statement does not return an error.

You must have the DBA privilege or be the owner of the table in order to create high or medium distributions.

Specifying High Distributions

If you use the HIGH keyword, the constructed distribution is exact, rather than statistically significant. Because of the time required to gather the information, you should use high distributions for specific tables or even columns rather than across the database. For very large tables, the database server may scan the data once for each column. The amount of space designated by the **DBUPSPACE** environment variable determines the number of times the table is scanned. For information about **DBUPSPACE**, see Chapter 4 of the *Informix Guide to SQL: Reference*.

If you do not specify a RESOLUTION clause, the default percentage is 0.5 percent.

Specifying MEDIUM Distributions

If you use the MEDIUM keyword, the data for the distributions is obtained by sampling. Because the data obtained by sampling is usually much smaller than the actual number of rows, the time required to construct medium distributions is less than the time required for high mode. Medium distributions require at least one scan of the table, so the creation of medium distributions executes more slowly than the creation of low distributions.

If you do not specify a RESOLUTION clause, the default percentage is 2.5 percent. If you do not specify a value for *conf*, the default confidence is 0.95. This can be roughly interpreted as meaning that 95 percent of the time, the estimate is equivalent to using high distributions.

Specifying DISTRIBUTIONS ONLY to Suppress Index Information

When you specify the MEDIUM or HIGH keywords, your UPDATE STATISTICS statement performs the functions of the LOW keyword as well. The functionality of the LOW keyword consists of constructing table information and index information for the specified objects. If you specify the DISTRIBUTIONS ONLY option with the MEDIUM or HIGH keywords, you prevent the construction of index information. However, table information is still constructed for the specified objects when you specify the DISTRIBUTIONS ONLY option. This information includes the number of pages used, the number of rows, and fragment information.

The reasons for suppressing index information but not table information with the DISTRIBUTIONS ONLY option are as follows:

- The table information is required to construct accurate distributions.
- The construction of index information can take a considerable amount of time, but the construction of table information requires very little time and very few system resources.

In the following example, the UPDATE STATISTICS statement gathers distributions information, index information, and table information for the **customer** table:

```
UPDATE STATISTICS MEDIUM FOR TABLE customer
```

However, in the following example, only distributions information and table information are gathered for the **customer** table. The DISTRIBUTIONS ONLY option prevents the construction of index information.

```
UPDATE STATISTICS MEDIUM FOR TABLE customer
    DISTRIBUTIONS ONLY
```

Recommended Procedure for Updating Statistics

Informix recommends the following procedure for giving the optimizer the best possible information while incurring the lowest performance penalty:

1. Run UPDATE STATISTICS in medium mode with the DISTRIBUTIONS ONLY option for each table. (If you are the database owner or DBA, and you want to gather statistics for the entire database, you can do that with a single command instead.). The default parameters are sufficient unless the table is very large. In this case, use a resolution of 1.00 and a confidence level of 0.99.
2. Run UPDATE STATISTICS in high mode for all columns that head an index. For the fastest execution time of the UPDATE STATISTICS statement, you must execute one UPDATE STATISTICS statement in the high mode for each such column. In a future release, Informix will remove this limitation.
3. For each multicolumn index, run UPDATE STATISTICS in low mode for all its columns.

This procedure executes rapidly because it only constructs the index-information statistics once for each index.

Update Statistics and Temporary Tables

You can use UPDATE STATISTICS on temporary tables. You can explicitly update the statistics for a temporary table or build distributions for a temporary table by specifying the name of the table. If you build distributions on all of the tables in the database by using the FOR TABLE clause without a specific table name, distributions will also be built on all the temporary tables in your session.

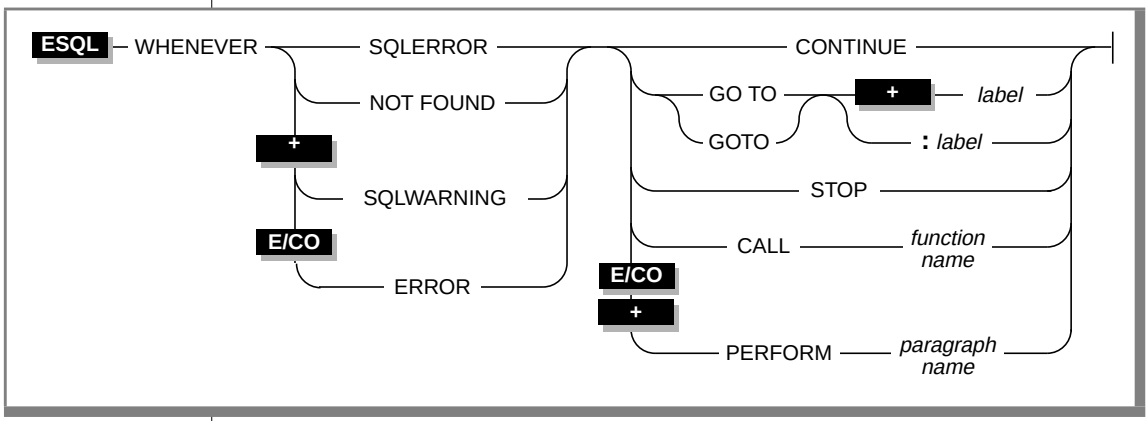
References

In the *INFORMIX-OnLine Dynamic Server Performance Guide*, see the discussion of UPDATE STATISTICS. In the *Informix Migration Guide*, see the discussion of how to use the **dbschema** utility to view distributions created with UPDATE STATISTICS.

WHENEVER

Use the WHENEVER statement to trap exceptions that occur during the execution of <vk>SQL statements.

Syntax



Element	Purpose	Restrictions	Syntax
function name	Function or procedure that is called when an exception occurs	Function or procedure must exist at compile time. A procedure in COBOL is equivalent to a function in C.	Function or procedure name must conform to language-specific rules for functions or procedures.
label	Statement label to which program control transfers when an exception occurs	An ESQL / COBOL label must be a paragraph name or a procedure name.	Label must conform to language-specific rules for statement labels.
paragraph name	Name of a paragraph of COBOL code to be executed when an SQL statement generates an error	The paragraph to be executed must exist within the program.	Paragraph name must conform to language-specific rules for paragraph names.

Type of Action	WHENEVER Keyword	For More Information
Transfer control to a specified label	GOTO GO TO	page 1-636
Transfer control to a named function or procedure	CALL	page 1-637
Transfer control to a specified COBOL paragraph	PERFORM (ESQL/COBOL only)	page 1-638

(2 of 2)

The Scope of WHENEVER

The ESQL preprocessor, not the database server, handles the interpretation of the WHENEVER statement. When the preprocessor encounters a WHENEVER statement in an ESQL source file, it inserts the appropriate code into the preprocessed code after each <vk>SQL statement based on the exception and the action that WHENEVER lists. The preprocessor defines the scope of a WHENEVER statement as from the point that it encounters the statement in the source module until it encounters one of the following conditions:

- The next WHENEVER statement with the same exception condition (SQLERROR, SQLWARNING, and NOT FOUND) in the same source module
- The end of the source module

Whichever condition the preprocessor encounters first as it sequentially processes the source module marks the end of the scope of the WHENEVER statement.

The following ESQL/C example program has three WHENEVER statements, two of which are WHENEVER SQLERROR statements. Line 4 uses STOP with SQLERROR to override the default CONTINUE action for errors. Line 8 specifies the CONTINUE keyword to return the handling of errors to the default behavior. For all <vk>SQL statements between lines 4 and 8, the preprocessor inserts code that checks for errors and halts program execution if an error occurs. Therefore, any errors that the INSERT statement on line 6 generates cause the program to stop.

After line 8, the preprocessor does not insert code to check for errors after <vk>SQL statements. Therefore, any errors that the INSERT statement (line 10), the SELECT statement (line 11), and DISCONNECT statement (line 12) generate are ignored. However, the SELECT statement does not stop program execution if it does not locate any rows; the WHENEVER statement on line 7 tells the program to continue if such an exception occurs.

```
1   main()
2   {

3   EXEC SQL connect to 'test';

4   EXEC SQL WHENEVER SQLERROR STOP;

5   printf("\n\nGoing to try first insert\n\n");
6   EXEC SQL insert into test_color values ('green');

7   EXEC SQL WHENEVER NOT FOUND CONTINUE;
8   EXEC SQL WHENEVER SQLERROR CONTINUE;

9   printf("\n\nGoing to try second insert\n\n");
10  EXEC SQL insert into test_color values ('blue');
11  EXEC SQL select paint_type from paint where color='red';
12  EXEC SQL disconnect all;
13  printf("\n\nProgram over\n\n");
14  }
```

SQLERROR Keyword

If you use the SQLERROR keyword, any <vk>SQL statement that encounters an error is handled as the WHENEVER SQLERROR statement directs. If an error occurs, the **sqlcode** variable is less than zero and the SQLSTATE variable has a class code with a value greater than 02. The following table lists the specification for the **sqlcode** variable for each product.

Product Name	Variable Name
ESQL / C	sqlca.sqlcode, SQLCODE
ESQL / COBOL	SQLCODE OF SQLCA

The following statement causes a program to stop execution if an <vk>SQL error exists:

```
WHENEVER SQLERROR STOP
```

If you do not use any WHENEVER SQLERROR statements in a program, the default for WHENEVER SQLERROR is CONTINUE.

SQLWARNING Keyword

If you use the SQLWARNING keyword, any <vk>SQL statement that generates a warning is handled as the WHENEVER SQLWARNING statement directs. If a warning occurs, the first field of the warning structure in SQLCA is set to W, and the SQLSTATE variable has a class code of 01. The following table lists the specification for the first field of the SQLCA warning structure for each product.

Product Name	Variable Name
ESQL / C	sqlca.sqlwarn.sqlwarn0
ESQL / COBOL	SQLWARN1 OF SQLWARN OF SQLCA

In addition to setting the first field of the warning structure, a warning also sets an additional field to W. The field that is set indicates the type of warning that occurred. For more information, see the chapter on exception checking in your <vk>SQL API manual.

The following statement causes a program to stop execution if a warning condition exists:

```
WHENEVER SQLWARNING STOP
```

If you do not use any WHENEVER SQLWARNING statements in a program, the default for WHENEVER SQLWARNING is CONTINUE.

NOT FOUND Keywords

If you use the NOT FOUND keywords, exception handling for SELECT and FETCH statements is treated differently than for other <vk>SQL statements. The NOT FOUND keyword checks for the following cases:

- The **End of Data** condition: a FETCH statement that attempts to get a row beyond the first or last row in the active set
- The **Not Found** condition: a SELECT statement that returns no rows

In each case, the **sqlcode** variable is set to 100, and the SQLSTATE variable has a class code of 02. For the name of the **sqlcode** variable in each Informix product, see the table in "SQLERROR Keyword" on page 1-634.

The following statement calls the **no_rows()** function each time the NOT FOUND condition exists:

```
WHENEVER NOT FOUND CALL no_rows
```

If you do not use any WHENEVER NOT FOUND statements in a program, the default for WHENEVER NOT FOUND is CONTINUE.

ERROR Keyword

E/CO

ERROR is a synonym for SQLERROR. For more information, see "SQLERROR Keyword" on page 1-634. ♦

CONTINUE Keyword

Use the CONTINUE keyword to instruct the program to ignore the exception and to continue execution at the next statement after the <vk>SQL statement. The default action for all exceptions is CONTINUE. You can use this keyword to turn off a previously specified option.

STOP Keyword

Use the STOP keyword to instruct the program to stop execution when the specified exception occurs. The following statement halts execution of an ESQL/C program each time that an <vk>SQL statement generates a warning:

```
EXEC SQL WHENEVER SQLWARNING STOP;
```

GOTO Keyword

Use the GOTO clause to transfer control to the statement that the label identifies when a particular exception occurs. The GOTO keyword is the ANSI-compliant syntax of the clause. The GO TO keywords are a non-ANSI synonym for GOTO.

The following example shows a WHENEVER statement in INFORMIX-ESQL / C code that transfers control to the label **missing** each time that the NOT FOUND condition occurs:

```
query_data()
    .
    .
    .
    EXEC SQL WHENEVER NOT FOUND GO TO missing;
    .
    .
    .
    EXEC SQL fetch lname into :lname;
    .
    .
    .
    missing:
        printf("No Customers Found\n");
    .
    .
    .
```

You must define the labeled statement in *each* program block that contains <vk>SQL statements. If your program contains more than one function, you might need to include the labeled statement and its code in *each* function. When the preprocessor reaches the function that does not contain the labeled statement, it tries to insert the code associated with the labeled statement. However, if you do not define this labeled statement within the function, the preprocessor generates an error.

To correct this error, either put a labeled statement with the same label name in each function, issue another WHENEVER statement to reset the error condition, or use the CALL clause to call a separate function.

CALL Clause

Use the CALL clause to transfer program control to the named function or procedure when a particular exception occurs. Do not include parentheses after the function or procedure name. The following WHENEVER statement causes the program to call the **error_recovery()** function if the program detects an error:

```
EXEC SQL WHENEVER SQLERROR CALL error_recovery;
```

When the named function completes, execution resumes at the next statement after the line that is causing the error. If you want to halt execution when an error occurs, include statements that terminate the program as part of the named function.

Observe the following restrictions on the named function:

- You cannot pass arguments to the named function nor can you return values from the named function. If the named function needs external information, use global variables or the GOTO clause of WHENEVER to transfer control to a label that calls the named function.
- You cannot specify the name of a stored procedure as a named function. To call a stored procedure, use the CALL clause to execute a function that contains the EXECUTE PROCEDURE statement.
- Make sure that all functions that the WHENEVER...CALL statement affects can find a declaration of the named function.

PERFORM Keyword for COBOL

Use the PERFORM keyword to execute a paragraph of COBOL code. The following example executes the COBOL paragraph ERR-CHK when an <vk>SQL statement generates an error:

```
EXEC SQL WHENEVER ERROR PERFORM ERR-CHK END-EXEC.
```

References

See the EXECUTE PROCEDURE and FETCH statements in this manual.

See the chapter on exception checking and error checking in your SQL API product manual.

Segments

Segments are language elements, such as table names and expressions, that occur repeatedly in the syntax diagrams for SQL and SPL statements. These language elements are discussed separately in this section for the sake of clarity, ease of use, and comprehensive treatment.

Whenever a segment occurs within the syntax diagram for an SQL or SPL statement, the diagram references the description of the segment in this section.

Scope of Segment Descriptions

The description of each segment includes the following information:

- A brief introduction that explains the purpose of the segment
- A syntax diagram that shows how to enter the segment correctly
- A syntax table that explains each input parameter in the syntax diagram
- Rules of usage, including examples that illustrate these rules

If a segment consists of multiple parts, the segment description provides the same set of information for each part. Each segment description concludes with references to related information in this manual and other manuals.

Use of Segment Descriptions

The syntax diagram within each segment description is not a standalone diagram. Instead it is a subdiagram that is subordinate to the syntax diagram for an SQL or SPL statement. A reference box in the syntax diagram for the statement refers to this subdiagram by providing the name of the segment and the page number on which the segment description begins.

First look up the syntax for the statement, and then turn to the segment description to find out the complete syntax for the segment. You will probably never need to look up the segment first and then work backward to a statement or statements that contain the segment.

For example, if you are using INFORMIX-OnLine Dynamic Server, and you want to enter a CREATE VIEW statement that includes a database name and database server name in the view name, first look up the syntax diagram for the CREATE VIEW statement. Then use the reference box for the View Name segment in that syntax diagram to look up the subdiagram for the View Name segment.

The subdiagram for the View Name segment shows you how to qualify the simple name of a view with the name of the database or with the name of both the database and the database server. Use the syntax in the subdiagram to enter a CREATE VIEW statement that includes the database name and database server name in the view name. The following example creates the **name_only** view in the **sales** database on the **boston** database server:

```
CREATE VIEW sales@boston:name_only AS
     SELECT customer_num, fname, lname FROM customer
```

Segments in This Section

This section describes the following segments:

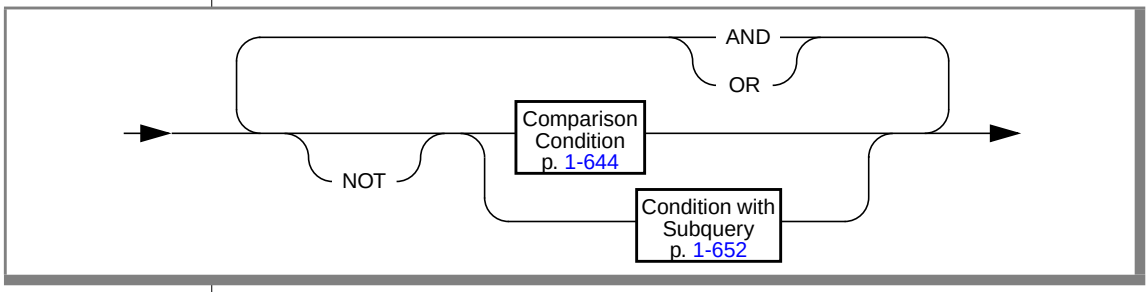
- Condition
- Constraint Name
- Database Name
- Data Type
- DATETIME Field Qualifier
- Expression
- Identifier
- Index Name
- INTERVAL Field Qualifier
- Literal DATETIME
- Literal INTERVAL
- Literal Number

- Procedure Name
- Quoted String
- Relational Operator
- Synonym Name
- Table Name
- View Name

Condition

A condition tests data to determine whether it meets certain qualifications. Use the Condition segment wherever you see a reference to a condition in a syntax diagram.

Syntax



Usage

A condition is a collection of one or more search conditions, optionally connected by the logical operators AND or OR. Search conditions fall into the following categories:

- Comparison conditions (also called filters or Boolean expressions)
- Conditions with a subquery

Restrictions on a Condition

A condition can contain only an aggregate function if it is used in the HAVING clause of a SELECT statement or the HAVING clause of a subquery. You cannot use an aggregate function in a comparison condition that is part of a WHERE clause in a DELETE, SELECT, or UPDATE statement unless the aggregate is on a correlated column that originates from a parent query and the WHERE clause is within a subquery that is within a HAVING clause.

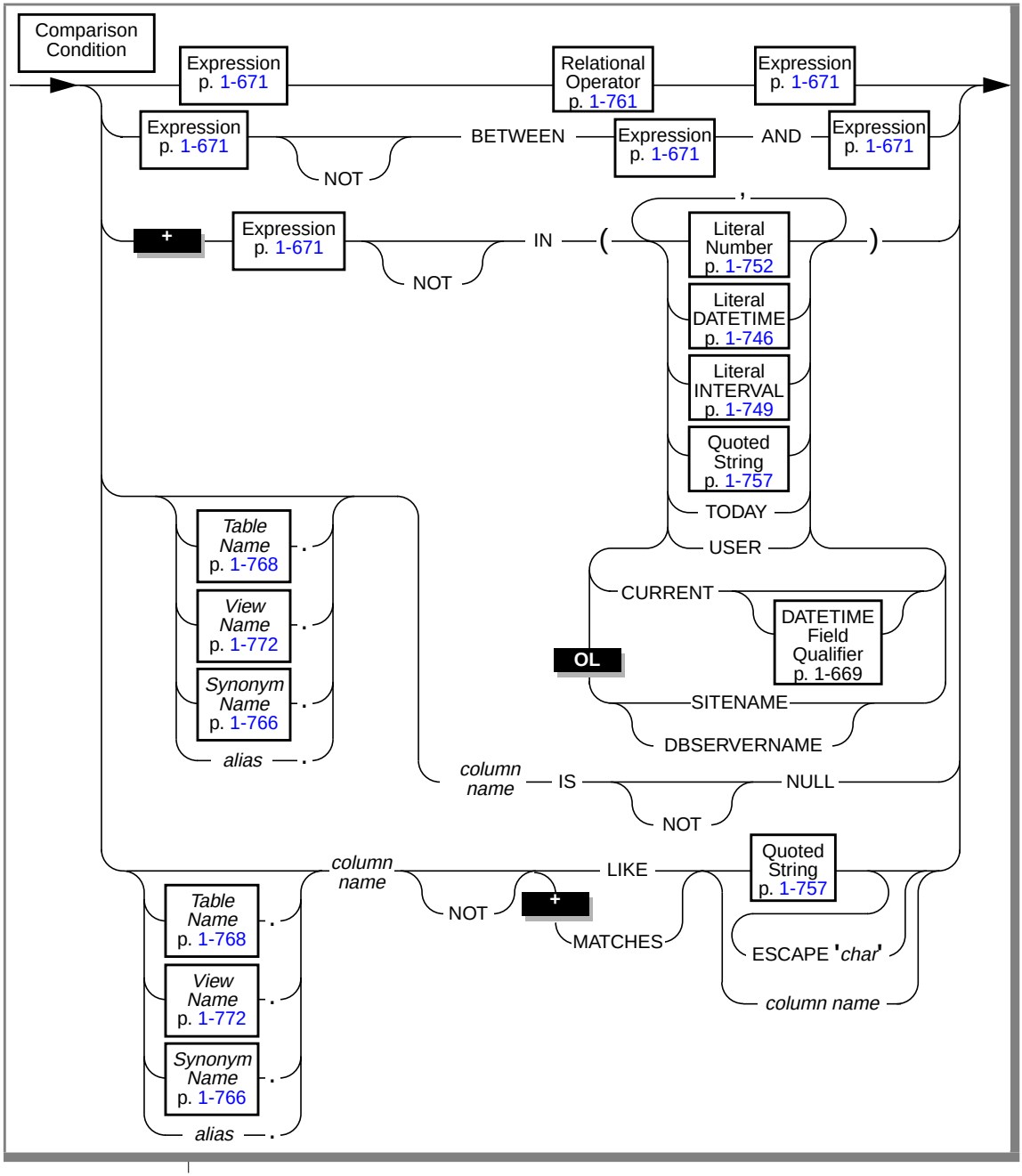
NOT Operator Option

If you preface a condition with the keyword NOT, the test is true only if the condition that NOT qualifies is false. If the condition that NOT qualifies is unknown (uses a null in the determination), the NOT operator has no effect. The following truth table shows the effect of NOT. The letter T represents a true condition, F represents a false condition, and a question mark (?) represents an unknown condition. Unknown values occur when part of an expression that uses an arithmetic operator is null.

NOT	
T	F
F	T
?	?

Comparison Conditions (Boolean Expressions)

Five kinds of comparison conditions exist: Relational Operator, BETWEEN, IN, IS NULL, and LIKE and MATCHES. Comparison conditions are often called Boolean expressions because they evaluate to a simple true or false result. Their syntax is summarized in the following diagram and explained in detail after the diagram.


Comparison Condition
Expression p. 1-671
Relational Operator p. 1-761
Expression p. 1-671
Expression p. 1-671
NOT
BETWEEN
Expression p. 1-671
AND
Expression p. 1-671
+
Expression p. 1-671
NOT
IN
(
,
)
Literal Number p. 1-752
Literal DATETIME p. 1-746
Literal INTERVAL p. 1-749
Quoted String p. 1-757
TODAY
USER
CURRENT
DATETIME Field Qualifier p. 1-669
OL
SITENAME
DBSERVERNAME
Table Name p. 1-768
View Name p. 1-772
Synonym Name p. 1-766
alias
.
column name
IS
NULL
NOT
column name
LIKE
NOT
+
MATCHES
Quoted String p. 1-757
ESCAPE 'char'
column name
Table Name p. 1-768
View Name p. 1-772
Synonym Name p. 1-766
alias

Element	Purpose	Restrictions	Syntax
alias	A temporary alternative name for a table or view within the scope of a SELECT statement	You must have defined the alias in the FROM clause of the SELECT statement.	Identifier, p. 1-723
char	A single ASCII character that is to be used as the escape character within the quoted string in a LIKE or MATCHES condition	See "ESCAPE with LIKE" on page 1-651 and "ESCAPE with MATCHES" on page 1-652.	Quoted String, p. 1-757
column name	The name of a column that is used in an IS NULL condition or in a LIKE or MATCHES condition. See "IS NULL Condition" on page 1-649 and "LIKE and MATCHES Condition" on page 1-649 for more information on the meaning of *column name* in these conditions.	The column must exist in the specified table.	Identifier, p. 1-723

Refer to the following sections for more information on the use of the different types of comparison conditions:

- For relational-operator conditions, refer to "Relational-Operator Condition" on page 1-647.
- For the BETWEEN condition, refer to "BETWEEN Condition" on page 1-648.
- For the IN condition, refer to "IN Condition" on page 1-648.
- For the IS NULL condition, refer to "IS NULL Condition" on page 1-649.
- For the LIKE and MATCHES condition, refer to "LIKE and MATCHES Condition" on page 1-649.

Quotation Marks in Conditions

When you compare a column expression with a constant expression in any type of comparison condition, observe the following rules:

- If the column has a numeric data type, you do not need to surround the constant expression with quotation marks.
- If the column has a character data type, you must surround the constant expression with quotation marks.
- If the column has a date data type, you should surround the constant expression with quotation marks. Otherwise, you might get unexpected results.

The following example shows the correct use of quotation marks in comparison conditions. The **ship_instruct** column has a character data type. The **order_date** column has a date data type. The **ship_weight** column has a numeric data type.

```
SELECT * FROM orders
    WHERE ship_instruct = 'express'
    AND order_date > '05/01/94'
    AND ship_weight < 30
```

Relational-Operator Condition

Some relational-operator conditions are shown in the following examples:

```
city[1,3] = 'San'

o.order_date > '6/12/86'

WEEKDAY(paid_date) = WEEKDAY(CURRENT-31 UNITS day)

YEAR(ship_date) < YEAR (TODAY)

quantity <= 3

customer_num <> 105

customer_num != 105
```

If either expression is null for a row, the condition evaluates to false. For example, if **paid_date** has a null value, you cannot use either of the following statements to retrieve that row:

```
SELECT customer_num, order_date FROM orders
    WHERE paid_date = ''

SELECT customer_num, order_date FROM orders
    WHERE NOT PAID !=''
```

An IS NULL condition finds a null value, as shown in the following example. The IS NULL condition is explained fully in "IS NULL Condition" on page 1-649.

```
SELECT customer_num, order_date FROM orders
    WHERE paid_date IS NULL
```

BETWEEN Condition

For a BETWEEN test to be true, the value of the expression on the left of the BETWEEN keyword must be in the inclusive range of the values of the two expressions on the right of the BETWEEN keyword. Null values do not satisfy the condition. You cannot use NULL for either expression that defines the range.

Some BETWEEN conditions are shown in the following examples:

```
order_date BETWEEN '6/1/93' and '9/7/93'

zipcode NOT BETWEEN '94100' and '94199'

EXTEND(call_dtime, DAY TO DAY) BETWEEN
    (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT

lead_time BETWEEN INTERVAL (1) DAY TO DAY
    AND INTERVAL (4) DAY TO DAY

unit_price BETWEEN loprice AND hiprice
```

IN Condition

The IN condition is satisfied when the expression to the left of the word IN is included in the list of items. The NOT option produces a search condition that is satisfied when the expression is not in the list of items. Null values do not satisfy the condition.

The following examples show some IN conditions:

```
WHERE state IN ('CA', 'WA', 'OR')

WHERE manu_code IN ('HRO', 'HSK')

WHERE user_id NOT IN (USER)

WHERE order_date NOT IN (TODAY)
```

ESQL

The TODAY function is evaluated at execution time; CURRENT is evaluated when a cursor opens or when the query executes, if it is a singleton SELECT statement. ♦

The USER function is case sensitive; it perceives **minnie** and **Minnie** as different values.

IS NULL Condition

The IS NULL condition is satisfied if the column contains a null value. If you use the IS NOT NULL option, the condition is satisfied when the column contains a value that is not null. The following example shows an IS NULL condition:

```
WHERE paid_date IS NULL
```

LIKE and MATCHES Condition

A LIKE or MATCHES condition tests for matching character strings. The condition is true, or satisfied, when either of the following tests is true:

- The value of the column on the left matches the pattern that the quoted string specifies. You can use wildcard characters in the string. Null values do not satisfy the condition.
- The value of the column on the left matches the pattern that the column on the right specifies. The value of the column on the right serves as the matching pattern in the condition.

You can use the single quote (') only with the quoted string to match a literal quote; you cannot use the ESCAPE keyword. You can use the quote character as the escape character in matching any other pattern if you write it as ''''.

NOT Option

The NOT option makes the search condition successful when the column on the left has a value that is not null and does not match the pattern that the quoted string specifies. For example, the following conditions exclude all rows that begin with the characters `Baxter` in the **lname** column:

```
WHERE lname NOT LIKE 'Baxter%'
WHERE lname NOT MATCHES 'Baxter*'
```

LIKE Option

If you use the keyword LIKE, you can use the following wildcard characters in the quoted string.

Wildcard	Meaning
%	The percent sign (%) matches zero or more characters.
_	The underscore (_) matches any single character.
\	The backslash (\) removes the special significance of the next character (used to match % or _ by writing \% or _).

Using the backslash (\) as an escape character is an Informix extension to ANSI-compliant SQL.

ANSI

If you use an escape character to escape anything other than percent sign (%), underscore (_), or the escape character itself, an error is returned. ♦

The following condition tests for the string `tennis`, alone or in a longer string, such as `tennis ball` or `table tennis paddle`:

```
WHERE description LIKE '%tennis%'
```

The following condition tests for all descriptions that contain an underscore. The backslash (\) is necessary because the underscore (_) is a wildcard character.

```
WHERE description LIKE '%\_%'
```

MATCHES Option

If you use the keyword MATCHES, you can use the following wildcard characters in the quoted string.

Wildcard	Meaning
*	The asterisk (*) matches zero or more characters.
?	The question mark (?) matches any single character.
[...]	The brackets ([...]) match any of the enclosed characters, including character ranges as in [a to z]. A caret (^) as the first character within the brackets matches any character that is not listed. Hence [^abc] matches any character that is not a, b, or c.
\	The backslash (\) removes the special significance of the next character (used to match * or ? by writing * or \?).

The following condition tests for the string `tennis`, alone or in a longer string, such as `tennis ball` or `table tennis paddle`:

```
WHERE description MATCHES '*tennis*'
```

The following condition is true for the names `Frank` and `frank`:

```
WHERE fname MATCHES '[Ff]rank'
```

The following condition is true for any name that begins with either `F` or `f`:

```
WHERE fname MATCHES '[Ff]*'
```

ESCAPE with LIKE

The ESCAPE clause lets you include an underscore (_) or a percent sign (%) in the quoted string and avoid having them be interpreted as wildcards. If you choose to use `z` as the escape character, the characters `z_` in a string stand for the character _. Similarly, the characters `z%` represent the percent sign (%). Finally, the characters `zz` in the string stand for the single character `z`. The following statement retrieves rows from the **customer** table in which the **company** column includes the underscore character:

```
SELECT * FROM customer
    WHERE company LIKE '%z_%' ESCAPE 'z'
```

You can also use a single-character host variable as an escape character. The following statement shows the use of a host variable as an escape character:

```
EXEC SQL BEGIN DECLARE SECTION;
    char escp='z';
    char fname[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL select fname from customer
    into :fname
    where company like '%z_%' escape :escp;
```

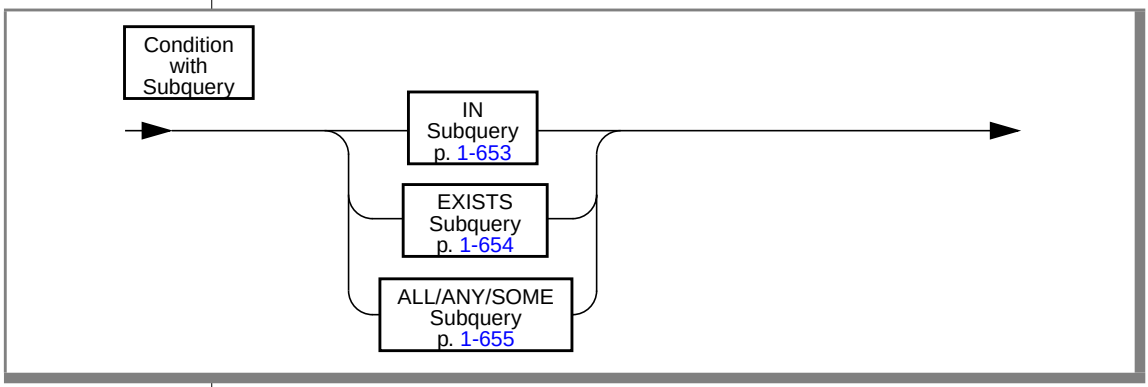
ESCAPE with MATCHES

The ESCAPE clause lets you include a question mark (?), an asterisk (*), and a left or right bracket ([]) in the quoted string and avoid having them be interpreted as wildcards. If you choose to use z as the escape character, the characters z? in a string stand for the question mark (?). Similarly, the characters z* stand for the asterisk (*). Finally, the characters zz in the string stand for the single character z.

The following example retrieves rows from the **customer** table in which the value of the **company** column includes the question mark (?):

```
SELECT * FROM customer
    WHERE company MATCHES '*z?*' ESCAPE 'z'
```

Condition with a Subquery



You can use a SELECT statement within a condition; this combination is called a subquery. You can use a subquery in a SELECT statement to perform the following functions:

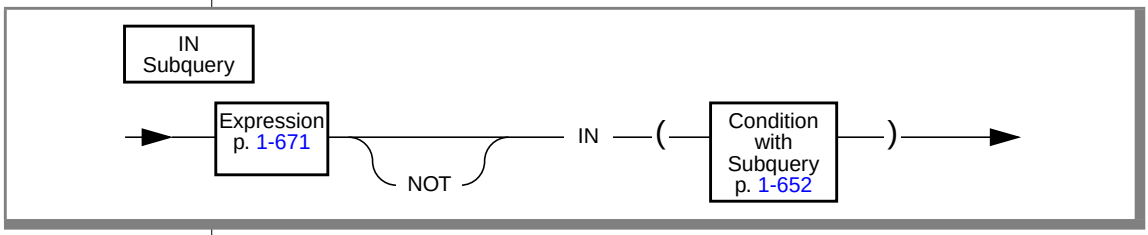
- Compare an expression to the result of another SELECT statement
- Determine whether an expression is included in the results of another SELECT statement
- Ask whether another SELECT statement selects any rows

The subquery can depend on the current row that the outer SELECT statement is evaluating; in this case, the subquery is a *correlated subquery*.

The kinds of subquery conditions are shown in the following sections with their syntax.

A subquery can return a single value, no value, or a set of values depending on the context in which it is used. If a subquery returns a value, it must select only a single column. If the subquery simply checks whether a row (or rows) exists, it can select any number of rows and columns. A subquery cannot contain an ORDER BY clause. The full syntax of the SELECT statement is described on page 1-459.

IN Subquery



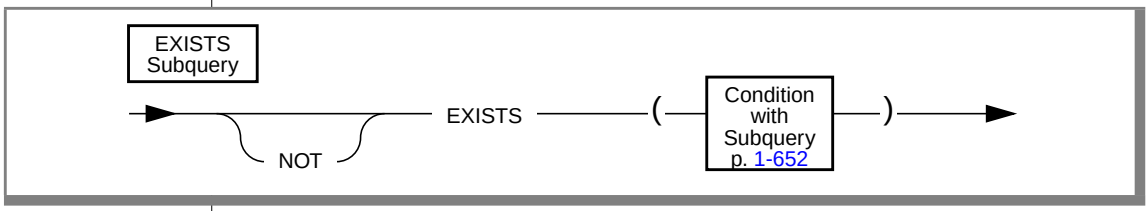
An IN subquery condition is true if the value of the expression matches one or more of the values that the subquery selects. The subquery must return only one column, but it can return more than one row. The keyword IN is equivalent to the =ANY sequence. The keywords NOT IN are equivalent to the !=ALL sequence. See "ALL / ANY / SOME Subquery" on page 1-655.

The following example of an IN subquery finds the order numbers for orders that do not include baseball gloves (**stock_num** = 1):

```
WHERE order_num NOT IN
    (SELECT order_num FROM items WHERE stock_num = 1)
```

Because the IN subquery tests for the presence of rows, duplicate rows in the subquery results do not affect the results of the main query. Therefore, you can put the UNIQUE or DISTINCT keyword into the subquery with no effect on the query results, although eliminating testing duplicates can reduce the time needed for running the query.

EXISTS Subquery



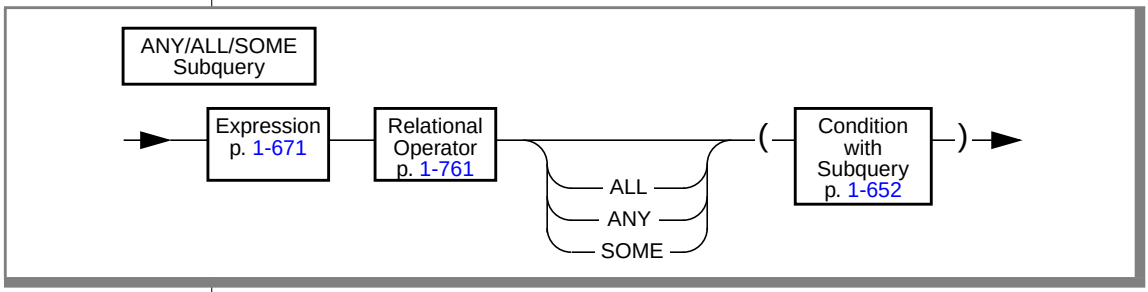
An EXISTS subquery condition evaluates to true if the subquery returns a row. With an EXISTS subquery, one or more columns can be returned. The subquery always contains a reference to a column of the table in the main query. If you use an aggregate function in an EXISTS subquery, at least one row is always returned.

The following example of a SELECT statement with an EXISTS subquery returns the stock number and manufacturer code for every item that has never been ordered (and is therefore not listed in the **items** table). You can appropriately use an EXISTS subquery in this SELECT statement because you use the subquery to test both **stock_num** and **manu_code** in **items**.

```
SELECT stock_num, manu_code FROM stock
    WHERE NOT EXISTS (SELECT stock_num, manu_code FROM items
        WHERE stock.stock_num = items.stock_num AND
        stock.manu_code = items.manu_code)
```

The preceding example works equally well if you use SELECT * in the subquery in place of the column names because the existence of the whole row is tested; specific column values are not tested.

ALL/ANY/SOME Subquery



You use the ALL, ANY, and SOME keywords to specify what makes the search condition true or false. A search condition that is true when the ANY keyword is used might not be true when the ALL keyword is used, and vice versa.

Keyword	Meaning
ALL	A keyword that denotes that the search condition is true if the comparison is true for every value that the subquery returns. If the subquery returns no value, the condition is true.
ANY	A keyword that denotes that the search condition is true if the comparison is true for at least one of the values that is returned. If the subquery returns no value, the search condition is false.
SOME	An alias for ANY

In the following example of the ALL subquery, the first condition tests whether each **total_price** is greater than the total price of every item in order number 1023. The second condition uses the MAX aggregate function to produce the same results.

```
total_price > ALL (SELECT total_price FROM items
                   WHERE order_num = 1023)

total_price > (SELECT MAX(total_price) FROM items
                   WHERE order_num = 1023)
```

The following conditions are true when the total price is greater than the total price of at least one of the items in order number 1023. The first condition uses the ANY keyword; the second uses the MIN aggregate function.

```
total_price > ANY (SELECT total_price FROM items
                   WHERE order_num = 1023)

total_price > (SELECT MIN(total_price) FROM items
                   WHERE order_num = 1023)
```

Using the NOT keyword with an ANY subquery tests whether an expression is not true for any subquery value. The condition, which is found in the following example of the NOT keyword with an ANY subquery, is true when the expression **total_price** is not greater than any selected value. That is, it is true when **total_price** is greater than none of the total prices in order number 1023.

```
NOT total_price > ANY (SELECT total_price FROM items
                       WHERE order_num = 1023)
```

Omitting ANY, ALL, or SOME Keywords

You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery will return exactly one value. If you omit the ANY, ALL, or SOME keywords, and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate function:

```
SELECT order_num FROM items
    WHERE stock_num = 9 AND quantity =
        (SELECT MAX(quantity) FROM items WHERE stock_num = 9)
```

Conditions with AND or OR

You can combine simple conditions with the logical operators AND or OR to form complex conditions. The following SELECT statements contain examples of complex conditions in their WHERE clauses:

```
SELECT customer_num, order_date FROM orders
    WHERE paid_date > '1/1/93' OR paid_date IS NULL

SELECT order_num, total_price FROM items
    WHERE total_price > 200.00 AND manu_code LIKE 'H%'

SELECT lname, customer_num FROM customer
    WHERE zipcode BETWEEN '93500' AND '95700'
    OR state NOT IN ('CA', 'WA', 'OR')
```

The following truth tables show the effect of the AND and OR operators.The letter T represents a true condition, F represents a false condition, and the question mark (?) represents an unknown value. Unknown values occur when part of an expression that uses a logical operator is null.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

If the Boolean expression evaluates to UNKNOWN, the condition is not satisfied.

Consider the following example within a WHERE clause:

```
WHERE ship_charge/ship_weight < 5
    AND order_num = 1023
```

The row where **order_num** = 1023 is the row where **ship_weight** is null. Because **ship_weight** is null, **ship_charge** / **ship_weight** is also null; therefore, the truth value of **ship_charge** / **ship_weight** < 5 is UNKNOWN. Because **order_num** = 1023 is TRUE, the AND table states that the truth value of the entire condition is UNKNOWN. Consequently, that row is not chosen. If the condition used an OR in place of the AND, the condition would be true.

References

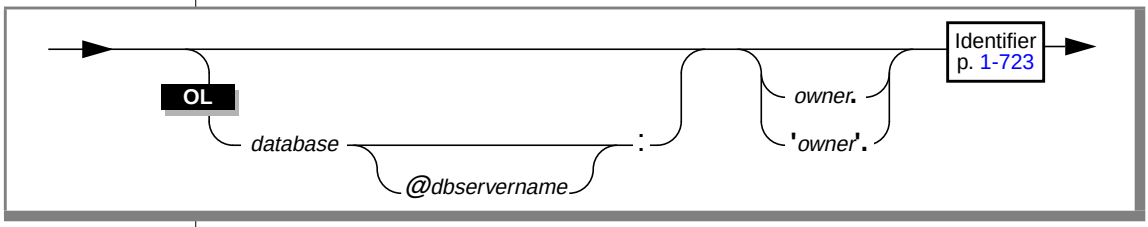
In the *Informix Guide to SQL: Tutorial*, see the discussion of conditions in the SELECT statement in Chapter 2 and Chapter 3.

In the *Guide to GLS Functionality*, see the discussion of the SELECT statement for information on the GLS aspects of conditions.

Constraint Name

The Constraint Name segment specifies the name of a constraint. Use the Constraint Name segment whenever you see a reference to a constraint name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the constraint resides	The database must exist.	Database Name, p. 1-660
dbservername	The name of the OnLine database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that *dbservername* specifies must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-660
owner	The user name of the owner of the constraint	If you are using an ANSI-compliant database, you must enter the *owner.* parameter for a constraint that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The actual name of the constraint is an <vk>SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of constraints. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

When you create a constraint, the name of the constraint must be unique within the database if the database is not ANSI compliant.

ANSI

When you create a constraint, the *owner.name* combination (the combination of the owner name and constraint name) must be unique within a database.

The *owner.name* combination is case sensitive. In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the constraint owner is stored as uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-770. ♦

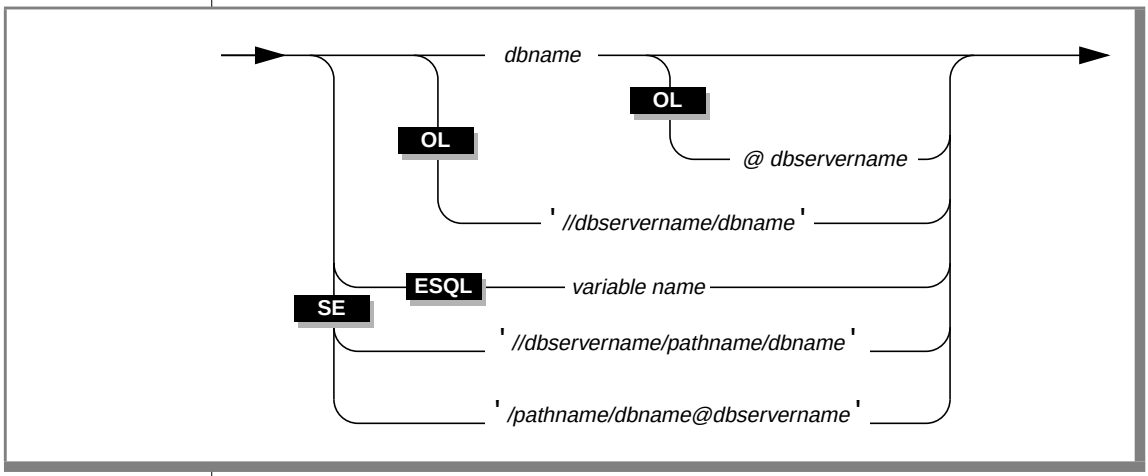
References

See the CREATE TABLE statement in this manual for information on defining constraints.

Database Name

The Database Name segment specifies the name of a database. Use the Database Name segment whenever you see a reference to a database name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dbname	The name of the database itself. This simple name does not include the pathname or the database server name.	A database name must be unique among the database names on the same database server. Database names are not case sensitive. If you are using OnLine, the database name can have a maximum of 18 characters. If you are using INFORMIX-SE, the database name can have a maximum of 10 characters. For restrictions on *dbname* that are specific to syntax formats that use quotation marks and slash symbols, see "/ / dbservername / dbname Option" on page 1-663 and "Specifying Server Names for INFORMIX-SE Databases" on page 1-663.	Identifier, p. 1-723
dbservername	The name of the database server on which the database that is named in *dbname* resides. The @ symbol is a literal character that introduces the database server name. Specifying a database server name allows you to choose a database on another database server as your current database. You can name the current database server using *dbservername*, although that is extra information.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file. You can put a space between *dbname* and the @ symbol, or you can omit the space. You cannot put a space between the @ symbol and *dbservername*. For restrictions on *dbservername* that are specific to syntax formats that use quotation marks and slash symbols, see "/ / dbservername / dbname Option" on page 1-663 and "Specifying Server Names for INFORMIX-SE Databases" on page 1-663.	Identifier, p. 1-723
pathname	The pathname of the database directory up to the parent directory of the **.dbs** directory	The specified path must exist. For punctuation restrictions on *pathname*, see "Specifying Server Names for INFORMIX-SE Databases" on page 1-663.	The pathname must conform to the conventions of your operating system.

(1 of 2)

Element	Purpose	Restrictions	Syntax
variable name	The name of a host variable that holds the name of a database	See "variable name Option" on page 1-663 for restrictions that apply when you are using INFORMIX-SE and you want to specify a database that does not reside either in your current directory or in a directory that the **DBPATH** environment variable specifies.	The name of the host variable must conform to language-specific rules for variable names.

(2 of 2)

Usage

The simple database name is an <vk>SQL identifier, as described on page 1-723. If you are creating a database, the name that you assign to the database can be 18 characters, inclusive. Database names are not case sensitive. You cannot use delimited identifiers for a database name.

SE

Database names in INFORMIX-SE databases can be 10 characters, inclusive. ♦

The maximum length of the database name and directory path, including *dbservername*, is 128 characters.

The following example shows a database specification:

```
empinfo@personnel
```

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of databases. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

@dbservername Option

If you use a database server name, do not put any spaces between the name and the @ symbol. For example, the following format is valid for the **stores7** database on the **training** database server:

```
stores7@training
```

//dbservername/dbname Option

If you use the alternative naming method, do not put spaces between the quotes, slashes, and names, as the following example shows:

```
'//training/stores7'
```

variable name Option

ESQL

You can use a variable within an SQL API to hold the name of a database. ♦

SE

ESQL

If you want to specify a database that resides neither in your current directory nor in a directory that the **DBPATH** environment variable specifies, you must specify a program variable that evaluates to the full pathname of the database (excluding the **.dbs** extension). ♦

Specifying Server Names for INFORMIX-SE Databases

SE

You can specify a database on a specific database server. Do not put spaces between the quotes, slashes, and names. The following database name describes a **stores7** database that resides on the **business** database server:

```
//business/u/acctng/demo/stores7
```

♦

SE

GLS

If you are using a nondefault locale that contains a multibyte code set, you must make sure that an SE database name meets the 10-byte size restriction. For more information, see the discussion on naming databases in Chapter 5 of the *Guide to GLS Functionality*. ♦

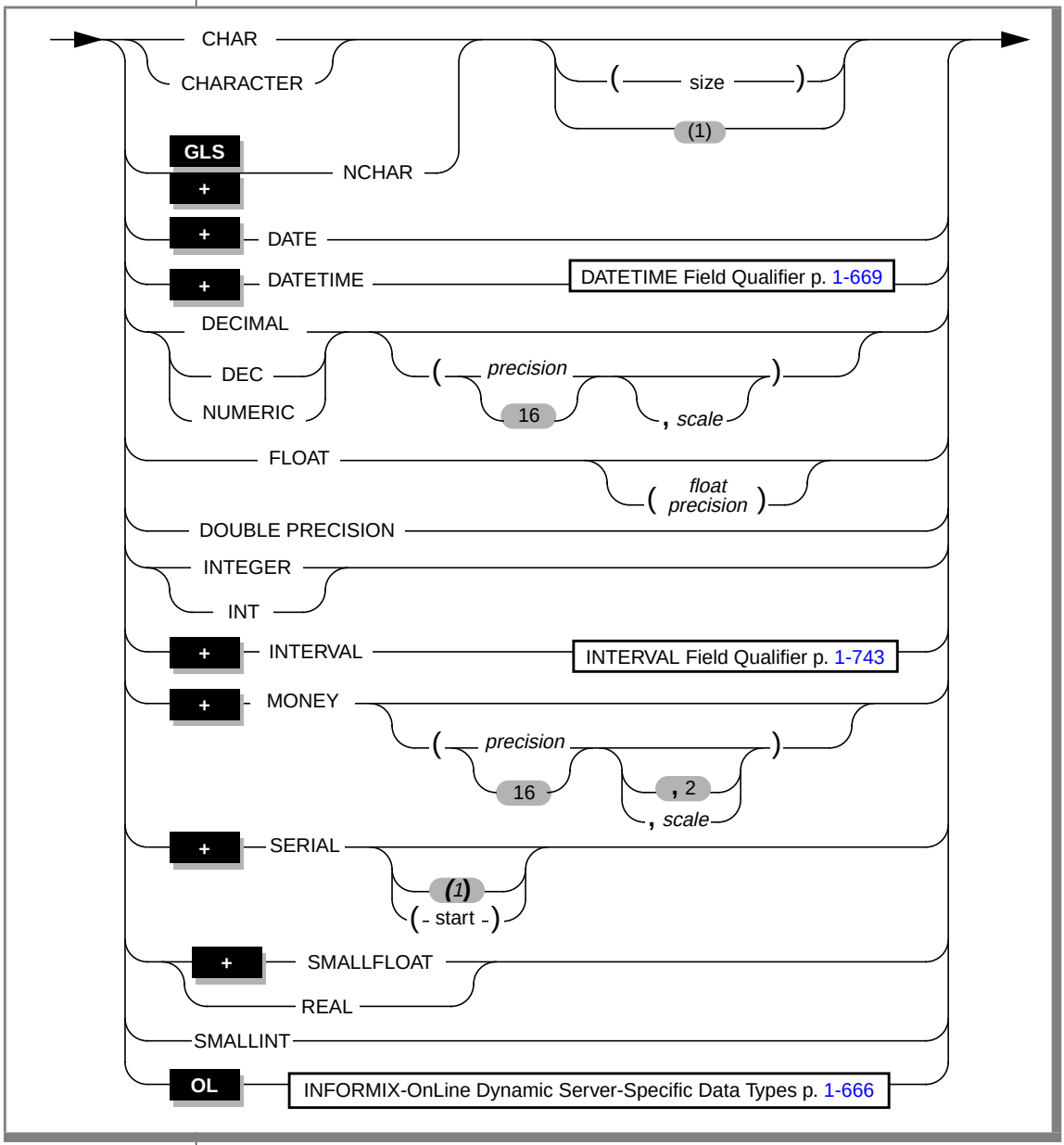
References

See the CREATE DATABASE and RENAME DATABASE statements in this manual for information on naming databases.

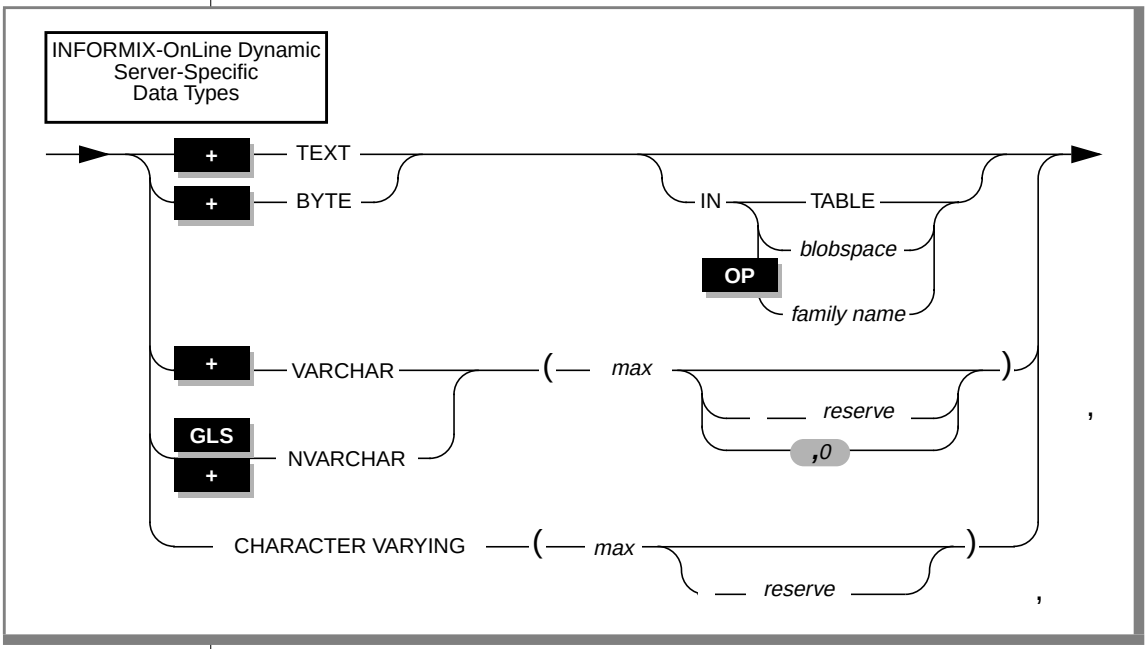
Data Type

The Data Type segment specifies the data type of a column or value. Use the Data Type segment whenever you see a reference to a data type in a syntax diagram.

Syntax


CHAR
CHARACTER
(size)
(1)
GLS
+
NCHAR
+
DATE
+
DATETIME
DATETIME Field Qualifier p. 1-669
DECIMAL
DEC
NUMERIC
(precision)
16
, scale
FLOAT
(float precision)
DOUBLE PRECISION
INTEGER
INT
+
INTERVAL
INTERVAL Field Qualifier p. 1-743
+
MONEY
(precision)
16
, 2
, scale
+
SERIAL
(1)
(- start -)
+
SMALLFLOAT
REAL
SMALLINT
OL
INFORMIX-OnLine Dynamic Server-Specific Data Types p. 1-666

INFORMIX-OnLine Dynamic Server-Specific Data Types



Element	Purpose	Restrictions	Syntax
blobspace	Name of an existing blobspace	The blobspace must exist.	Identifier, p. 1-723
family name	Quoted string constant that specifies a family name or variable name in the optical family	The family name or variable name must exist.	Quoted String, p. 1-757 For additional information about optical families, see *INFORMIX-OnLine/Optical User Manual*.
float precision	The float precision is ignored.	You must specify a positive integer.	Literal Number, p. 1-752
max	Maximum size of a CHARACTER VARYING or VARCHAR or NVARCHAR column in bytes	You must specify an integer value between 1 and 255 bytes inclusive. If you place an index on the column, the largest value you can specify for *max* is 254 bytes.	Literal Number, p. 1-752

(1 of 2)

Element	Purpose	Restrictions	Syntax
precision	Total number of significant digits in a decimal or money data type	You must specify an integer between 1 and 32, inclusive.	Literal Number, p. 1-752
reserve	Amount of space in bytes reserved for a CHARACTER VARYING or VARCHAR or NVARCHAR column even if the actual number of bytes stored in the column is less than *reserve*	You must specify an integer value between 0 and 255 bytes. However, the value you specify for *reserve* must be less than the value you specify for *max*.	Literal Number, p. 1-752
scale	Number of digits to the right of the decimal point	You must specify an integer between 1 and *precision*.	Literal Number, p. 1-752
size	Number of bytes in the CHAR or NCHAR column.	For OnLine, you must specify an integer value between 1 and 32,767 bytes inclusive. For SE, you must specify an integer value between 1 and 32,511 bytes inclusive.	Literal Number, p. 1-752
start	Starting number for values in a SERIAL column	You must specify an integer greater than 0 and less than 2,147,483,647.	Literal Number, p. 1-752

(2 of 2)

For more information, see the discussion of all data types in Chapter 3 of the *Informix Guide to SQL: Reference*.

Fixed and Varying Length Data Types

The data type CHAR is for fixed-length character data. Use the ANSI-compliant CHARACTER VARYING data type to specify varying length character data. You can also specify varying length data with the Informix VARCHAR data type.

NCHAR and NVARCHAR Data Types

GLS

See Chapter 3 of the *Guide to GLS Functionality* for a discussion of the NCHAR and NVARCHAR data types. ♦

References

In the *Informix Guide to SQL: Reference*, see the discussions of individual data types in Chapter 3.

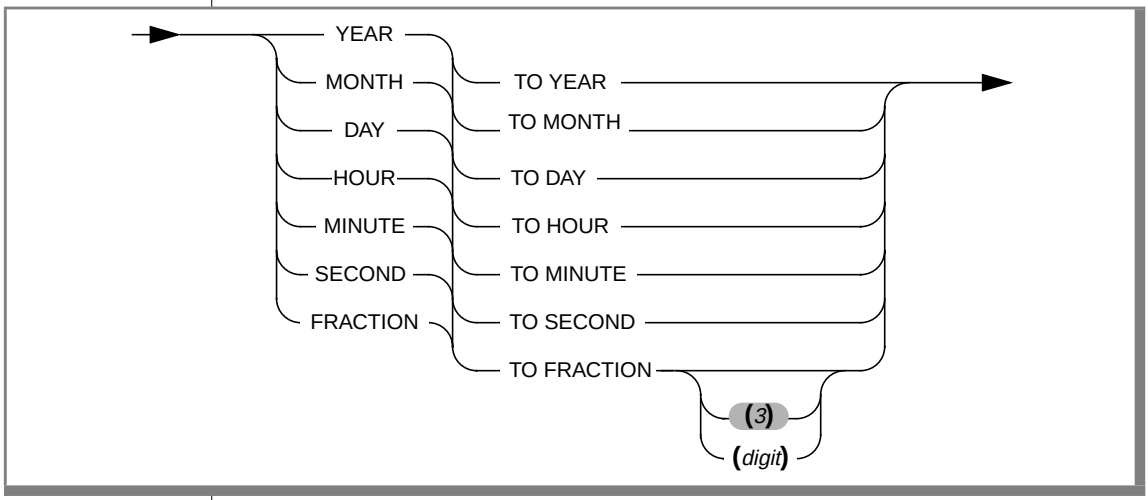
In the *Informix Guide to SQL: Tutorial*, see the discussion of data types in Chapter 9.

In the *Guide to GLS Functionality*, see the discussion of the NCHAR and NVARCHAR data types and the GLS aspects of other character data types.

DATETIME Field Qualifier

A DATETIME field qualifier specifies the largest and smallest unit of time in a DATETIME column or value. Use the DATETIME Field Qualifier segment whenever you see a reference to a DATETIME field qualifier in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
digit	A single integer that specifies the precision of a decimal fraction of a second. The default precision is 3 digits (a thousandth of a second).	You must specify an integer between 1 and 5, inclusive.	Literal Number, p. 1-752

Usage

Specify the largest unit for the first DATETIME value; after the word TO, specify the smallest unit for the value. The keywords imply that the following values are used in the DATETIME object.

Unit of Time	Meaning
YEAR	Specifies a year, numbered from A.D. 1 to 9999
MONTH	Specifies a month, numbered from 1 to 12
DAY	Specifies a day, numbered from 1 to 31, as appropriate to the month in question
HOUR	Specifies an hour, numbered from 0 (midnight) to 23
MINUTE	Specifies a minute, numbered from 0 to 59
SECOND	Specifies a second, numbered from 0 to 59
FRACTION	Specifies a fraction of a second, with up to five decimal places. The default scale is three digits (thousandth of a second).

The following examples show DATETIME qualifiers:

```
DAY TO MINUTE

YEAR TO MINUTE

DAY TO FRACTION(4)

MONTH TO MONTH
```

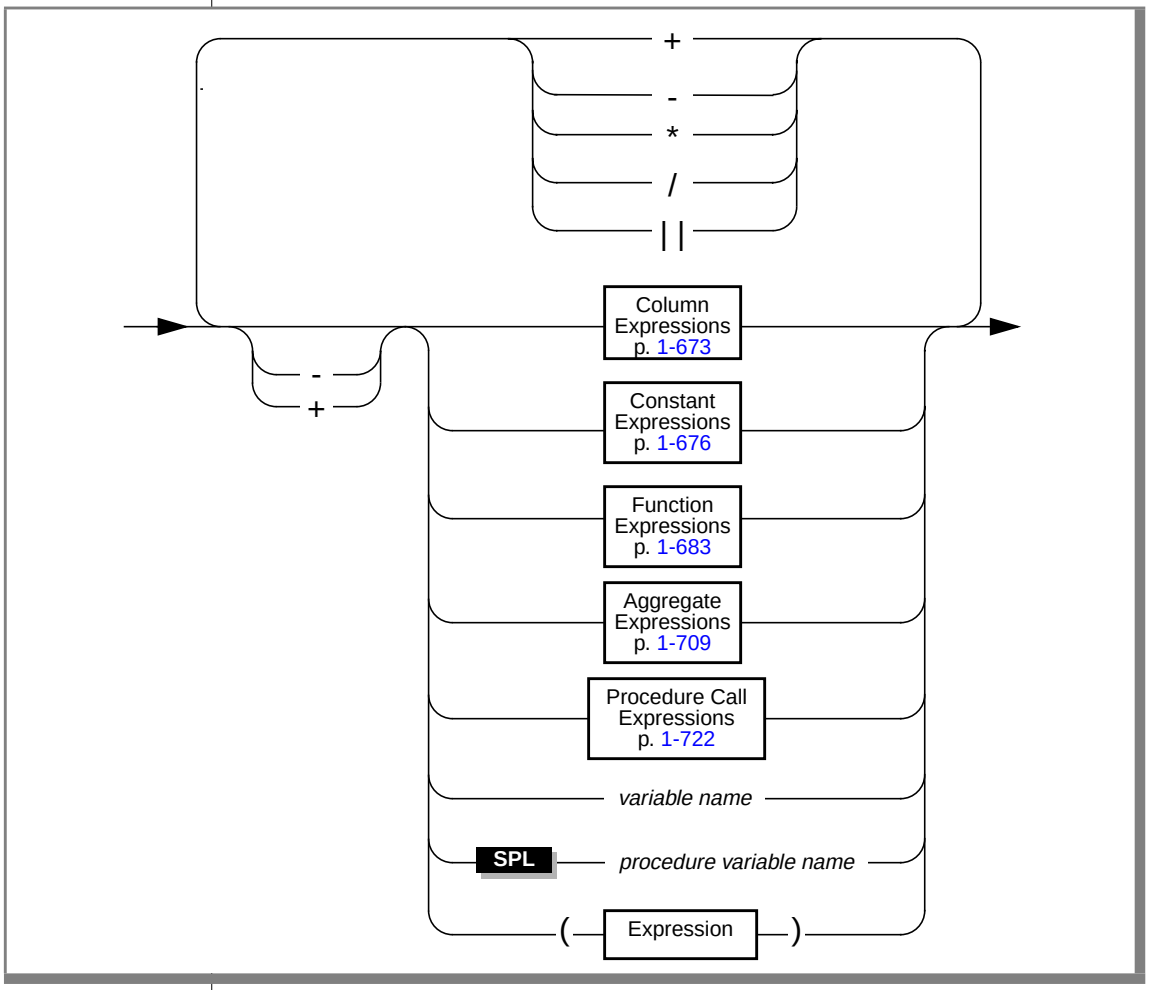
References

In the *Informix Guide to SQL: Reference*, see the DATETIME data type in Chapter 3 for an explanation of the DATETIME field qualifier.

Expression

An expression is one or more pieces of data that is contained in or derived from the database or database server. Use the Expression segment whenever you see a reference to an expression in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
procedure variable name	The name of a variable that is defined in a stored procedure. The value stored in the variable is one of the expression types that is shown in the syntax diagram.	The expression that is stored in *procedure variable name* must conform to the rules for expressions of that type.	Identifier, p. 1-723
variable name	The name of a program variable or host variable. The value stored in the variable is one of the expression types shown in the syntax diagram.	The expression that is stored in *variable name* must conform to the rules for expressions of that type.	The name of the variable must conform to language-specific rules for variable names.

Usage

To combine expressions, connect them with arithmetic operators for addition, subtraction, multiplication, and division.

You cannot use an aggregate expression in a condition that is part of a WHERE clause unless the aggregate expression is used within a subquery.

Concatenation Operator

You can use the concatenation operator (||) to concatenate two expressions. For example, the following examples are some possible concatenated-expression combinations. The first example concatenates the **zipcode** column to the first three letters of the **lname** column. The second example concatenates the suffix **.dbg** to the contents of a host variable called **file_variable**. The third example concatenates the value returned by the TODAY function to the string **Date**.

```
lname[1,3] || zipcode

:file_variable || '.dbg'

'Date:' || TODAY
```

ESQL

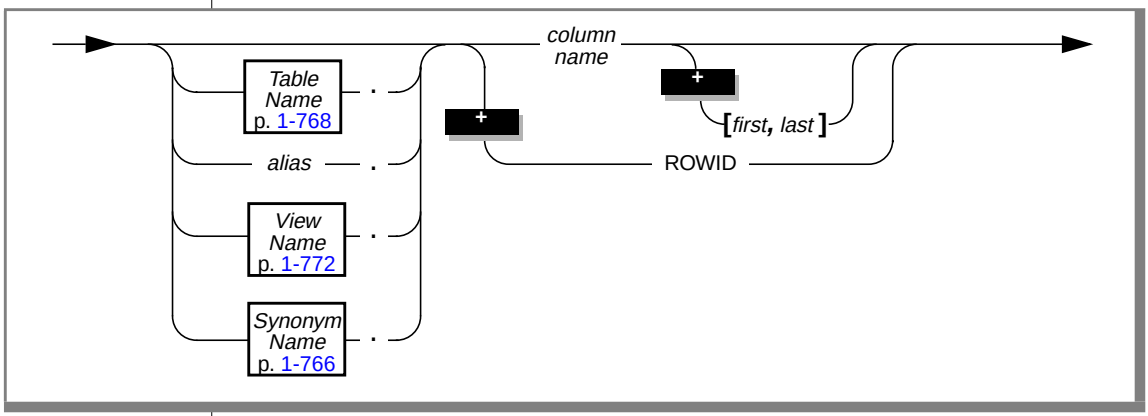
You cannot use the concatenation operator in an embedded-language-only statement. The SQL API-only statements appear in the following list.

ALLOCATE DESCRIPTOR	FETCH
CLOSE	FLUSH
CONNECT TO	FREE
DEALLOCATE DESCRIPTOR	GET DESCRIPTOR
DECLARE	OPEN
DESCRIBE	PREPARE
DISCONNECT	PUT
EXECUTE	SET CONNECTION
EXECUTE IMMEDIATE	SET DESCRIPTOR

♦

Column Expressions

The possible syntax for column expressions is shown in the following diagram.



Element	Purpose	Restrictions	Syntax
alias	A temporary alternative name for a table or view within the scope of a SELECT statement. This alternative name is established in the FROM clause of the SELECT statement.	The restrictions depend on the clause of the SELECT statement in which *alias* occurs.	Identifier, p. 1-723
column name	The name of the column that you are specifying	The restrictions depend on the statement in which *column name* occurs.	Identifier, p. 1-723
first	The position of the first character in the portion of the column that you are selecting	The column must be one of the following types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-752
last	The position of the last character in the portion of the column that you are selecting	The column must be one of the following types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.	Literal Number, p. 1-752

The following examples show column expressions:

```
company

items.price

cat_advert [1,15]
```

Use a table or alias name whenever it is necessary to distinguish between columns that have the same name but are in different tables. The SELECT statements that the following example shows use **customer_num** from the **customer** and **orders** tables. The first example precedes the column names with table names. The second example precedes the column names with table aliases.

```
SELECT * FROM customer, orders
    WHERE customer.customer_num = orders.customer_num

SELECT * FROM customer c, orders o
    WHERE c.customer_num = o.customer_num
```

Using Subscripts on Character Columns

You can use subscripts on **CHAR**, **VARCHAR**, **NCHAR**, **NVARCHAR**, **BYTE**, and **TEXT** columns. The subscripts indicate the starting and ending character positions that are contained in the expression. Together the column subscripts define a column substring. The column substring is the portion of the column that is contained in the expression.

For example, if a value in the **lname** column of the **customer** table is `Greenburg`, the following expression evaluates to `burg`:

```
lname[6,9]
```

GLS

For information on the GLS aspects of column subscripts and substrings, see the *Guide to GLS Functionality*, Chapter 3. ♦

Using Rowids

You can use the rowid column that is associated with a table row as a property of the row. The rowid column is essentially a hidden column in nonfragmented tables and in fragmented tables that were created with the WITH ROWIDS clause. The rowid column is unique for each row, but it is not necessarily sequential. Informix recommends, however, that you utilize primary keys as an access method rather than exploiting the **rowid** column.

SE

The rowid column is sequential and starts at 1 for each table. ♦

The following examples show possible uses of the ROWID keyword in a SELECT statement:

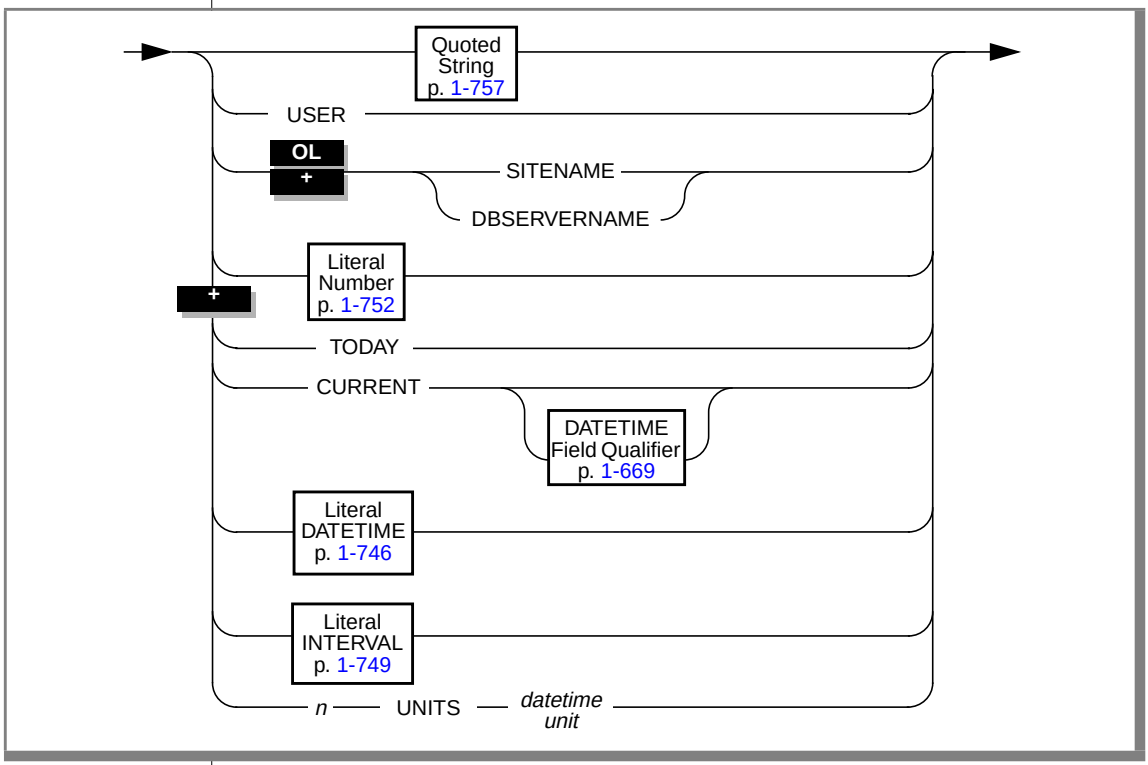
```
SELECT *, ROWID FROM customer

SELECT fname, ROWID FROM customer
ORDER BY ROWID

SELECT HEX(rowid) FROM customer
WHERE customer_num = 106
```

In INFORMIX-OnLine Dynamic Server only, the last SELECT statement example shows how to get the page number (the first six digits after 0x) and the slot number (the last two digits) of the location of your row.

You cannot use ROWID keyword in the select list of a query that contains an aggregate function.

Constant Expressions

The following diagram shows the possible syntax for constant expressions.



- For the CURRENT function in an expression, see "CURRENT Function" on page 1-680.
- For literal DATETIME as an expression, see "Literal DATETIME as an Expression" on page 1-681.
- For literal INTERVAL as an expression, see "Literal INTERVAL as an Expression" on page 1-682.
- For the UNITS keyword in an expression, see "UNITS Keyword" on page 1-682.

Quoted String as an Expression

The following examples show quoted strings as expressions:

```
SELECT 'The first name is ', fname FROM customer

INSERT INTO manufact VALUES ('SPS', 'SuperSport')

UPDATE cust_calls SET res_dtime = '1993-1-1 10:45'
    WHERE customer_num = 120 AND call_code = 'B'
```

USER Function

The USER function returns a string that contains the login name of the current user (that is, the person running the process).

The following statements show how you might use the USER function:

```
INSERT INTO cust_calls VALUES
   (221,CURRENT,USER,'B','Decimal point off', NULL, NULL)

SELECT * FROM cust_calls WHERE user_id = USER

UPDATE cust_calls SET user_id = USER WHERE customer_num = 220
```

The USER function does not change the case of a user ID. If you use USER in an expression and the present user is **Robertm**, the USER function returns **Robertm**, not **robertm**. If you specify user as the default value for a column, the column must be CHAR, VARCHAR, NCHAR, or NVARCHAR data type, and it must be at least eight characters long.

ANSI

In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the table owner is stored as uppercase letters. If you use the USER keyword as part of a condition, you must be sure that the way the user name is stored agrees with the values that the USER function returns, with respect to case. ♦

SITENAME and DBSERVERNAME Functions

The SITENAME and DBSERVERNAME functions return the database server name, as defined in the ONCONFIG file for the INFORMIX-OnLine Dynamic Server installation where the current database resides or as specified in the **INFORMIXSERVER** environment variable. The two function names, SITENAME and DBSERVERNAME, are synonymous. You can use the DBSERVERNAME function to determine the location of a table, to put information into a table, or to extract information from a table. You can insert DBSERVERNAME into a simple character field or use it as a default value for a column. If you specify DBSERVERNAME as a default value for a column, the column must be CHAR, VARCHAR, NCHAR, or NVARCHAR data type and must be at least 18 characters long.

In the following example, the first statement returns the name of the database server where the **customer** table resides. Because the query is not restricted with a WHERE clause, it returns DBSERVERNAME for every row in the table. If you add the DISTINCT keyword to the SELECT clause, the query returns DBSERVERNAME once. The second statement adds a row that contains the current site name to a table. The third statement returns all the rows that have the site name of the current system in **site_col**. The last statement changes the company name in the **customer** table to the current system name.

```
SELECT DBSERVERNAME FROM customer

INSERT INTO host_tab VALUES ('1', DBSERVERNAME)

SELECT * FROM host_tab WHERE site_col = DBSERVERNAME

UPDATE customer SET company = DBSERVERNAME
     WHERE customer_num = 120
```

Literal Number as an Expression

The following examples show literal numbers as expressions:

```
INSERT INTO items VALUES (4, 35, 52, 'HRO', 12, 4.00)

INSERT INTO acreage VALUES (4, 5.2e4)

SELECT unit_price + 5 FROM stock

SELECT -1 * balance FROM accounts
```

TODAY Function

Use the TODAY function to return the system date as a DATE data type. If you specify TODAY as a default value for a column, it must be a DATE column.

The following examples show how you might use the TODAY function in an INSERT, UPDATE, or SELECT statement:

```
UPDATE orders (order_date) SET order_date = TODAY
    WHERE order_num = 1005

INSERT INTO orders VALUES
   (0, TODAY, 120, NULL, N, '1AUE217', NULL, NULL, NULL, NULL)

SELECT * FROM orders WHERE ship_date = TODAY
```

CURRENT Function

The CURRENT function returns a DATETIME value with the date and time of day, showing the current instant.

If you do not specify a datetime qualifier, the default qualifiers are YEAR TO FRACTION(3). You can use the CURRENT function in any context in which you can use a literal DATETIME (see page 1-746). If you specify CURRENT as the default value for a column, it must be a DATETIME column and the qualifier of CURRENT must match the column qualifier, as the following example shows:

```
CREATE TABLE new_acct (col1 int, col2 DATETIME YEAR TO DAY
    DEFAULT CURRENT YEAR TO DAY)
```

If you use the CURRENT keyword in more than one place in a single statement, identical values can be returned at each point of the call. You cannot rely on the CURRENT function to provide distinct values each time it executes.

The returned value comes from the system clock and is fixed when any SQL statement starts. For example, any calls to CURRENT from an EXECUTE PROCEDURE statement return the value when the stored procedure starts.

The CURRENT function is always evaluated in the database server where the current database is located. If the current database is in a remote database server, the returned value is from the remote host.

The CURRENT function might not execute in the physical order in which it appears in a statement. You should not use the CURRENT function to mark the start, end, or a specific point in the execution of a statement.

If your platform does not provide a system call that returns the current time with subsecond precision, the CURRENT function returns a zero for the FRACTION field.

In the following example, the first statement uses the CURRENT function in a WHERE condition. The second statement uses the CURRENT function as the input for the DAY function. The last query selects rows whose **call_dtime** value is within a range from the beginning of 1993 to the current instant.

```
DELETE FROM cust_calls WHERE
    res_dtime < CURRENT YEAR TO MINUTE

SELECT * FROM orders WHERE DAY(ord_date) < DAY(CURRENT)

SELECT * FROM cust_calls WHERE call_dtime
    BETWEEN '1993-1-1 00:00:00' AND CURRENT
```

Literal DATETIME as an Expression

The following examples show literal DATETIME as an expression:

```
SELECT DATETIME (1993-12-6) YEAR TO DAY FROM customer

UPDATE cust_calls SET res_dtime = DATETIME (1992-07-07 10:40)
          YEAR TO MINUTE
    WHERE customer_num = 110
    AND call_dtime = DATETIME (1992-07-07 10:24) YEAR TO MINUTE

SELECT * FROM cust_calls
    WHERE call_dtime
    = DATETIME (1995-12-25 00:00:00) YEAR TO SECOND
```

Literal INTERVAL as an Expression

The following examples show literal INTERVAL as an expression:

```
INSERT INTO manufact VALUES ('CAT', 'Catwalk Sports',
    INTERVAL (16) DAY TO DAY)

SELECT lead_time + INTERVAL (5) DAY TO DAY FROM manufact
```

The second statement in the preceding example adds five days to each value of **lead_time** selected from the **manufact** table.

UNITS Keyword

The UNITS keyword enables you to display a simple interval or increase or decrease a specific interval or datetime value.

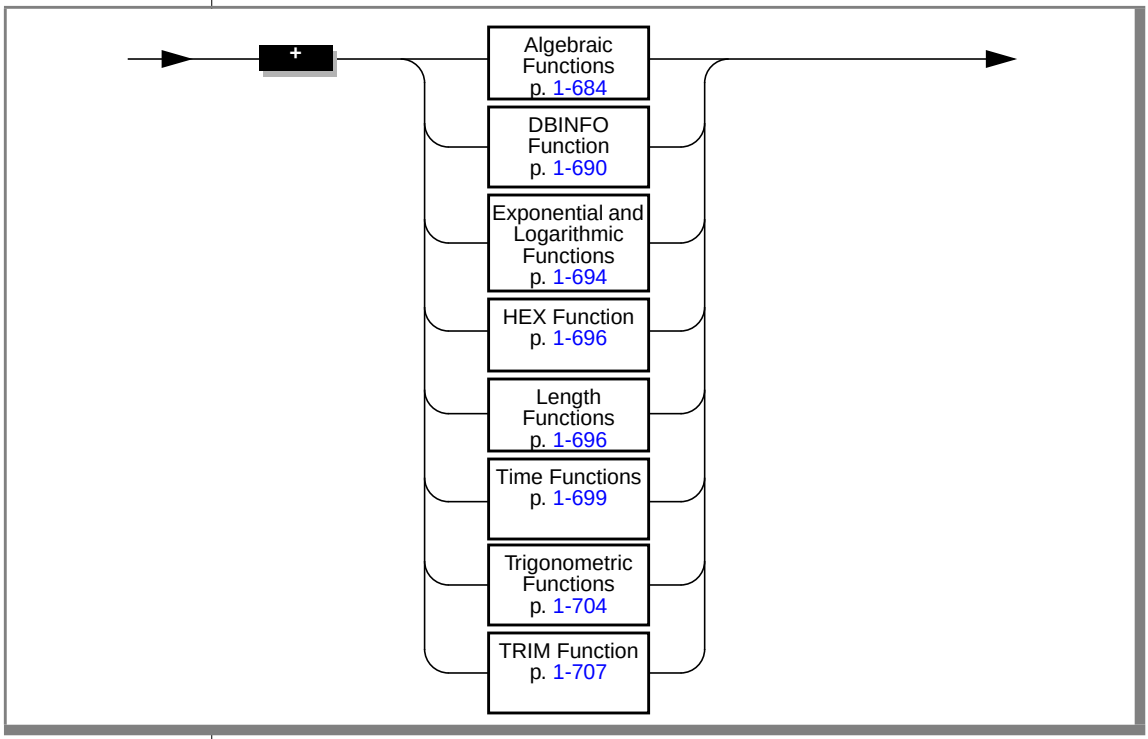
If *n* is not an integer, it is rounded down to the nearest whole number when it is used.

In the following example, the first SELECT statement uses the UNITS keyword to select all the manufacturer lead times, increased by five days. The second SELECT statement finds all the calls that were placed more than 30 days ago. If the expression in the WHERE clause returns a value greater than 99 (maximum number of days), the query fails. The last statement increases the lead time for the ANZA manufacturer by two days.

```
SELECT lead_time + 5 UNITS DAY FROM manufact

SELECT * FROM cust_calls
    WHERE (TODAY - call_dtime) > 30 UNITS DAY

UPDATE manufact SET lead_time = 2 UNITS DAY + lead_time
    WHERE manu_code = 'ANZ'
```

Function Expressions

A function expression takes an argument, as the following diagram shows.



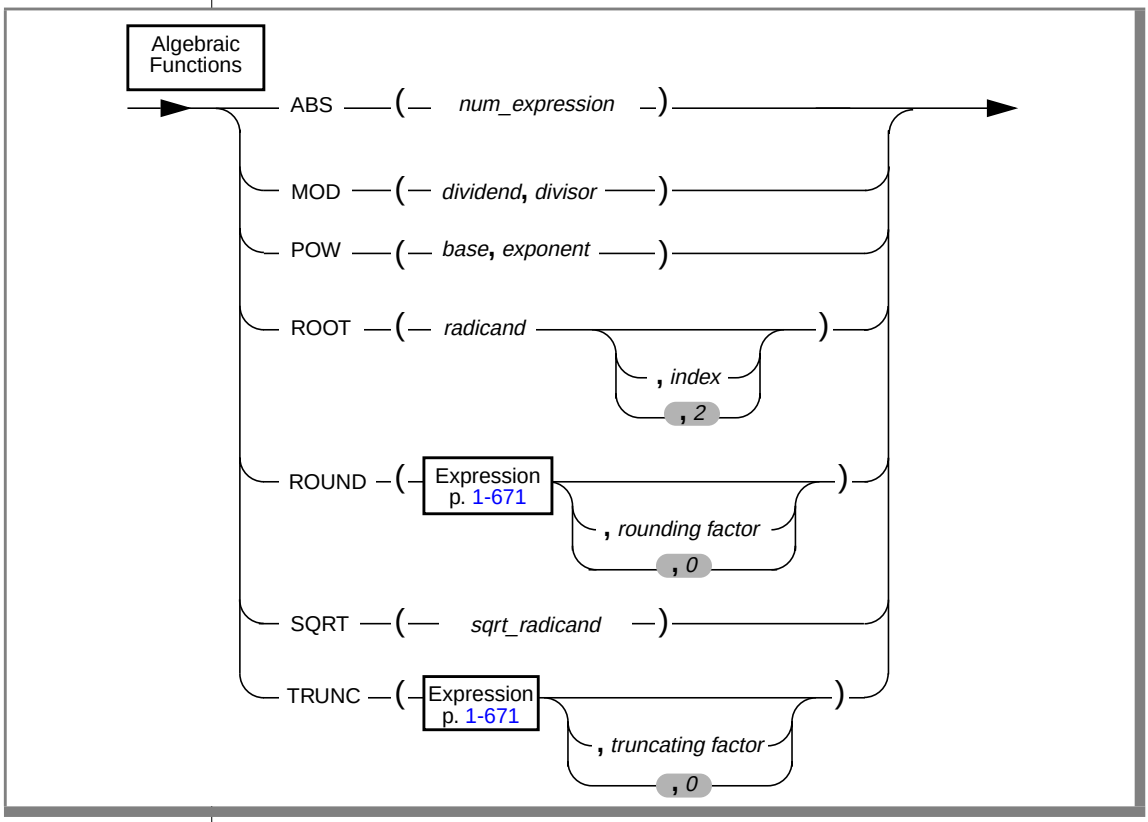
The following examples show function expressions:

```
EXTEND (call_dtime, YEAR TO SECOND)

MDY (12, 7, 1900 + cur_yr)

DATE (365/2)

LENGTH ('abc') + LENGTH (pvar)

HEX (customer_num)

HEX (LENGTH(123))

TAN (radians)
```

```
ABS (-32)
EXP (4,3)
MOD (10,3)
```

Algebraic Functions

An algebraic function takes one or more arguments, as the following diagram shows.



Element	Purpose	Restrictions	Syntax
base	A value to be raised to the power that is specified in *exponent*. The *base* value is the first argument that is supplied to the POW function. See "POW Function" on page 1-688 for further information on *base*.	You can enter in *base* any real number or any expression that evaluates to a real number.	Expression, p. 1-671
dividend	A value to be divided by the value in *divisor*. The *dividend* value is the first argument supplied to the MOD function. See "MOD Function" on page 1-687 for further information on *dividend*.	You can enter in *dividend* any real number or any expression that evaluates to a real number.	Expression, p. 1-671
divisor	The value by which the value in *dividend* is to be divided. The *divisor* value is the second argument that is supplied to the MOD function. See "MOD Function" on page 1-687 for further information on *divisor*	You can enter in *divisor* any real number except zero or any expression that evaluates to a real number other than zero.	Expression, p. 1-671
exponent	The power to which the value that is specified in *base* is to be raised. The *exponent* value is the second argument that is supplied to the POW function. See "POW Function" on page 1-688 for further information on *exponent*.	You can enter in *exponent* any real number or any expression that evaluates to a real number.	Expression, p. 1-671
index	The type of root to be returned, where 2 represents square root, 3 represents cube root, and so on. The *index* value is the second argument that is supplied to the ROOT function. The default value of *index* is 2. See "ROOT Function" on page 1-688 for further information on *index*.	You can enter in *index* any real number except zero or any expression that evaluates to a real number other than zero.	Expression, p. 1-671

(1 of 3)

Element	Purpose	Restrictions	Syntax
num_expression	A numeric expression for which an absolute value is to be returned. The expression serves as the argument for the ABS function. See "ABS Function" on page 1-687 for further information on *num_expression*.	The value of *num_expression* can be any real number.	Expression, p. 1-671
radicand	An expression whose root value is to be returned. The *radicand* value is the first argument that is supplied to the ROOT function. See "ROOT Function" on page 1-688 for further information on *radicand*.	You can enter in *radicand* any real number or any expression that evaluates to a real number.	Expression, p. 1-671
rounding factor	The number of digits to which a numeric expression is to be rounded. The *rounding factor* value is the second argument that is supplied to the ROUND function. The default value of *rounding factor* is zero. This default means that the numeric expression is rounded to zero digits or the ones place. See "ROUND Function" on page 1-688 for further information on *rounding factor*.	The value you specify in *rounding factor* must be an integer between +32 and -32, inclusive. See "ROUND Function" on page 1-688 for further information on this restriction.	Literal Number, p. 1-752
sqrt_radicand	An expression whose square root value is to be returned. The *sqrt_radicand* value is the argument that is supplied to the SQRT function. See "SQRT Function" on page 1-689 for further information on *sqrt_radicand*.	You can enter in *sqrt_radicand* any real number or any expression that evaluates to a real number.	Expression, p. 1-671

(2 of 3)

Element	Purpose	Restrictions	Syntax
truncating factor	The position to which a numeric expression is to be truncated. The *truncating factor* value is the second argument that is supplied to the TRUNC function.The default value of *truncating factor* is zero. This default means that the numeric expression is truncated to zero digits or the ones place. See "TRUNC Function" on page 1-690 for further information on *truncating factor.*	The value you specify in *truncating factor* must be an integer between +32 and -32, inclusive. See "TRUNC Function" on page 1-690 for further information on this restriction.	Literal Number, p. 1-752

(3 of 3)

ABS Function

The ABS function gives the absolute value for a given expression. The function requires a single numeric argument. The value returned is the same as the argument type. The following example shows all orders of more than $20 paid in cash (+) or store credit (-). The **stores7** database does not contain any negative balances; however, you might have negative balances in your application.

```
SELECT order_num, customer_num, ship_charge
  FROM orders WHERE ABS(ship_charge) > 20
```

MOD Function

The MOD function returns the modulus or remainder value for two numeric expressions. You provide integer expressions for the dividend and divisor. The divisor cannot be 0. The value returned is INT. The following example uses a 30-day billing cycle to determine how far into the billing cycle today is:

```
SELECT MOD(today - MDY(1,1,year(today)),30) FROM orders
```

POW Function

The POW function raises the *base* to the *exponent*. This function requires two numeric arguments. The return type is FLOAT. The following example returns all the information for circles whose areas (πr^2) are less than 1,000 square units:

```
SELECT * FROM circles WHERE (3.1417 * POW(radius,2)) < 1000
```

ROOT Function

The ROOT function returns the root value of a numeric expression. This function requires at least one numeric argument (the *radicand* argument) and allows no more than two (the *radicand* and *index* arguments). If only the *radicand* argument is supplied, the value 2 is used as a default value for the *index* argument. The value 0 cannot be used as the value of *index*. The value that the ROOT function returns is FLOAT. The first SELECT statement in the following example takes the square root of the expression. The second SELECT statement takes the cube root of the expression.

```
SELECT ROOT(9) FROM angles          -- square root of 9

SELECT ROOT(64,3) FROM angles       -- cube root of 64
```

The SQRT function uses the form SQRT(x)=ROOT(x) if no index is given.

ROUND Function

The ROUND function returns the rounded value of an expression. The expression must be numeric or must be converted to numeric.

If you omit the digit indication, the value is rounded to zero digits or to the ones place. The digit limitation of 32 (+ and -) refers to the entire decimal value.

Positive-digit values indicate rounding to the right of the decimal point; negative-digit values indicate rounding to the left of the decimal point, as Figure 1-4 shows.

Figure 1-4
ROUND Function

The following example shows how you can use the ROUND function with a column expression in a SELECT statement. This statement displays the order number and rounded total price (to zero places) of items whose rounded total price (to zero places) is equal to 124.00.

```
SELECT order_num , ROUND(total_price) FROM items
    WHERE ROUND(total_price) = 124.00
```

If you use a MONEY data type as the argument for the ROUND function and you round to zero places, the value displays with .00. The SELECT statement in the following example rounds an INTEGER value and a MONEY value. It displays 125 and a rounded price in the form xxx.00 for each row in **items**.

```
SELECT ROUND(125.46), ROUND(total_price) FROM items
```

SQRT Function

The SQRT function returns the square root of a numeric expression.

The following example returns the square root of 9 for each row of the **angles** table:

```
SELECT SQRT(9) FROM angles
```

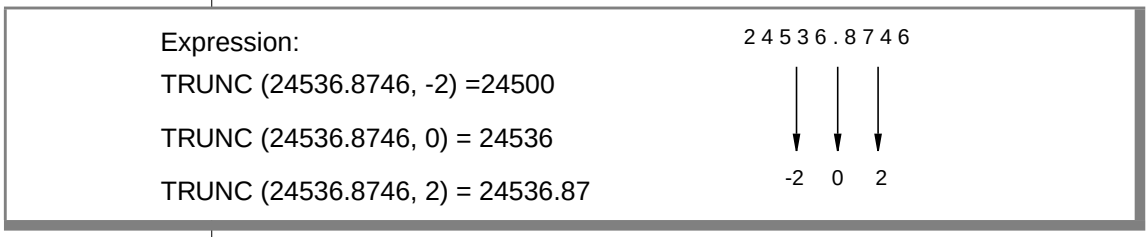
TRUNC Function

The TRUNC function returns the truncated value of a numeric expression.

The expression must be numeric or a form that can be converted to a numeric expression. If you omit the digit indication, the value is truncated to zero digits or to the one's place. The digit limitation of 32 (+ and -) refers to the entire decimal value.

Positive digit values indicate truncating to the right of the decimal point; negative digit values indicate truncating to the left of the decimal point, as Figure 1-5 shows.

Figure 1-5
TRUNC Function



If you use a MONEY data type as the argument for the TRUNC function and you truncate to zero places, the .00 places are removed. For example, the following SELECT statement truncates a MONEY value and an INTEGER value. It displays 125 and a truncated price in integer format for each row in **items**.

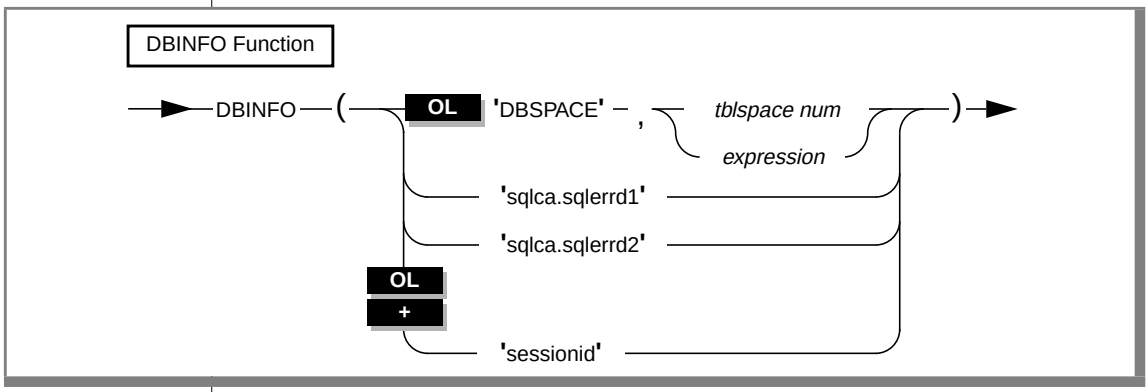
```
SELECT TRUNC(125.46), TRUNC(total_price) FROM items
```

DBINFO Function

Use the DBINFO function for any of the following purposes:

- To locate the name of a dbspace corresponding to a tblspace number or expression
- To find out the last serial value inserted in a table
- To find out the number of rows processed by selects, inserts, deletes, updates, and execute procedure statements
- To find out the session ID of the current session

You can use the DBINFO function anywhere within SQL statements and within stored procedures.



Element	Purpose	Restrictions	Syntax
expression	An expression that evaluates to *tblspace num*	The expression can contain procedure variables, host variables, column names, or subqueries, but it must evaluate to a numeric value.	Expression, p. 1-671
tblspace num	The tblspace number (partition number) of a table. The DBSPACE option of the DBINFO function returns the name of the dbspace that corresponds to the specified tblspace number.	The specified tblspace number must exist. That is, it must occur in the **partnum** column of the **systables** table for the database.	Literal Number, p. 1-752

Using the 'DBSPACE' Option

The 'DBSPACE' option returns a character string that contains the name of the dbspace corresponding to a tblspace number. You must supply an additional parameter, either *tblspace num* or an expression that evaluates to *tblspace num*. The following example uses the 'DBSPACE' option. First, it queries the **systables** system catalog table to determine the *tblspace num* for the table **customer**, then it executes the function to determine the dbspace name.

```
SELECT tabname, partnum FROM systables;
```

If the statement returns a partition number of 16777289, you insert that value into the second argument to find which dbspace contains the **customer** table, as shown in the following example:

```
SELECT DBINFO ('DBSPACE', 16777289) FROM systables;
```

Using the 'sqlca.sqlerrd1' Option

The 'sqlca.sqlerrd1' option returns a single integer that provides the last serial value that is inserted into a table. To ensure valid results, use this option immediately following an INSERT statement that inserts a serial value. The following example uses the 'sqlca.sqlerrd1' option:

```
.
.
EXEC SQL create table fst_tab (ordernum serial, partnum int);
EXEC SQL create table sec_tab (ordernum serial);

EXEC SQL insert into fst_tab VALUES (0,1);
EXEC SQL insert into fst_tab VALUES (0,4);
EXEC SQL insert into fst_tab VALUES (0,6);

EXEC SQL insert into sec_tab
    select dbinfo('sqlca.sqlerrd1')
    from sec_tab where partnum = 6;
.
.
```

This example inserts a row that contains a primary-key serial value into the **fst_tab** table, and then uses the DBINFO() function to insert the same serial value into the **sec_tab** table. The value that the DBINFO() function returns is the serial value of the last row that is inserted into **fst_tab**. The subquery in the last line contains a WHERE clause so that a single value is returned.

Using the 'sqlca.sqlerrd2' Option

The 'sqlca.sqlerrd2' option returns a single integer that provides the number of rows that SELECT, INSERT, DELETE, UPDATE, and EXECUTE PROCEDURE statements processed. To ensure valid results, use this option after SELECT and EXECUTE PROCEDURE statements have completed executing. In addition, if you use this option within cursors, make sure that all rows are fetched before the cursors are closed to ensure valid results.

The following example shows a stored procedure that uses the 'sqlca.sqlerrd2' option to determine the number of rows that are deleted from a table:

```
CREATE PROCEDURE del_rows (pnumb int)
RETURNING int;

DEFINE nrows int;

DELETE FROM sec_tab WHERE partnum=pnumb;
LET nrows = DBINFO('sqlca.sqlerrd2');
RETURN nrows;

END PROCEDURE
```

Using the 'sessionid' Option

The 'sessionid' option of the DBINFO function returns the session ID of your current session.

When a client application makes a connection to INFORMIX-OnLine Dynamic Server, the database server starts a session with the client and assigns a session ID for the client. The session ID serves as a unique identifier for a given connection between a client and a database server. The database server stores the value of the session ID in a data structure in shared memory that is called the session control block. The session control block for a given session also includes the user ID, the process ID of the client, the name of the host computer, and a variety of status flags.

When you specify the 'sessionid' option, the database server retrieves the session ID of your current session from the session control block and returns this value to you as an integer. Some of the System-Monitoring Interface (SMI) tables in the **sysmaster** database include a column for session IDs, so you can use the session ID that the DBINFO function obtained to extract information about your own session from these SMI tables. For further information on the session control block, the **sysmaster** database, and the SMI tables, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide*.

In the following example, the user specifies the DBINFO function in a SELECT statement to obtain the value of the current session ID. The user poses this query against the **systables** system catalog table and uses a WHERE clause to limit the query result to a single row.

```
SELECT DBINFO('sessionid') AS my_sessionid
    FROM systables
    WHERE tabname = 'systables'
```

The following table shows the result of this query.

my_sessionid
14

In the preceding example, the SELECT statement queries against the **systables** system catalog table. However, you can obtain the session ID of the current session by querying against any system catalog table or user table in the database. For example, you can enter the following query to obtain the session ID of your current session:

```
SELECT DBINFO('sessionid') AS user_sessionid
    FROM customer
    where customer_num = 101
```

The following table shows the result of this query.

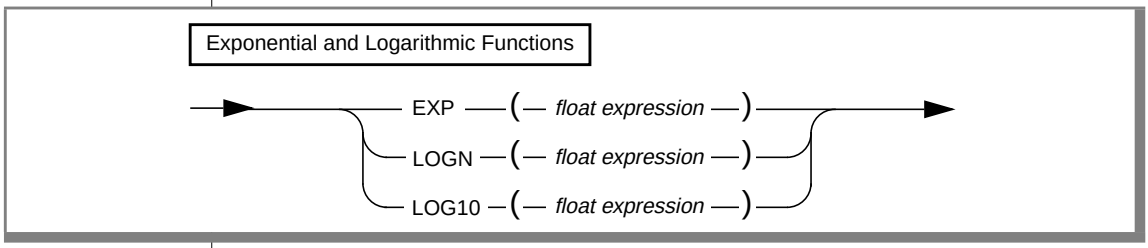
user_sessionid
14

You can use the DBINFO function not only in SQL statements but also in stored procedures. The following example shows a stored procedure that returns the value of the current session ID to the calling program or procedure:

```
CREATE PROCEDURE get_sess()
RETURNING INT;
RETURN DBINFO('sessionid');
END PROCEDURE;
```

Exponential and Logarithmic Functions

Exponential and logarithmic functions take at least one argument. The return type is FLOAT. The following example shows exponential and logarithmic functions.



Element	Purpose	Restrictions	Syntax
float expression	An expression that serves as an argument to the EXP, LOGN, or LOG10 functions. For information on the meaning of *float expression* in these functions, see the individual heading for each function on the following pages.	The domain of the expression is the set of real numbers, and the range of the expression is the set of positive real numbers.	Expression, p. 1-671

EXP Function

The EXP function returns the exponential value of two numeric expressions. You provide a constant and float expression in the form `e(n)=en`. The following example returns the exponent of 3 for each row of the **angles** table:

```
SELECT EXP(3) FROM angles
```

LOGN Function

The LOGN function returns the natural log of a numeric expression. The logarithmic value is the inverse of the exponential value. The following SELECT statement returns the natural log of population for each row of the **history** table:

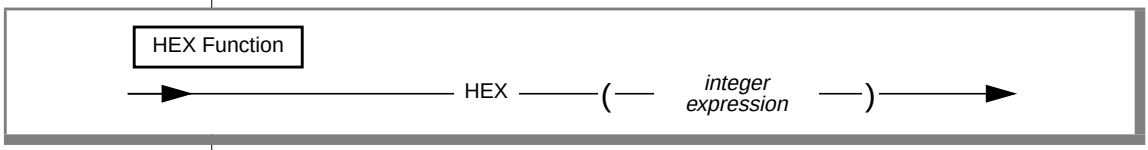
```
SELECT LOGN(population) FROM history WHERE country='US'
    ORDER BY date
```

LOG10 Function

The LOG10 function returns the log of a value to the base 10. The following example returns the log base 10 of distance for each row of the **travel** table:

```
SELECT LOG10(distance) + 1 digits FROM travel
```

HEX Function



Element	Purpose	Restrictions	Syntax
integer expression	A numeric expression for which you want to know the hexadecimal equivalent	You must specify an integer or an expression that evaluates to an integer.	Expression, p. 1-671

The HEX function returns the hexadecimal encoding of an integer expression. The following example displays the data type and column length of the columns of the **orders** table in hexadecimal format. For MONEY and DECIMAL columns, you can then determine the precision and scale from the lowest and next-to-the-lowest bytes. For VARCHAR and NVARCHAR columns, you can determine the minimum space and maximum space from the lowest and next to the lowest bytes. (See Chapter 2 of the *Informix Guide to SQL: Reference* for more information about encoded information.)

```
SELECT colname, coltype, HEX(collength)
    FROM syscolumns C, systables T
    WHERE C.tabid = T.tabid AND T.tabname = 'orders'
```

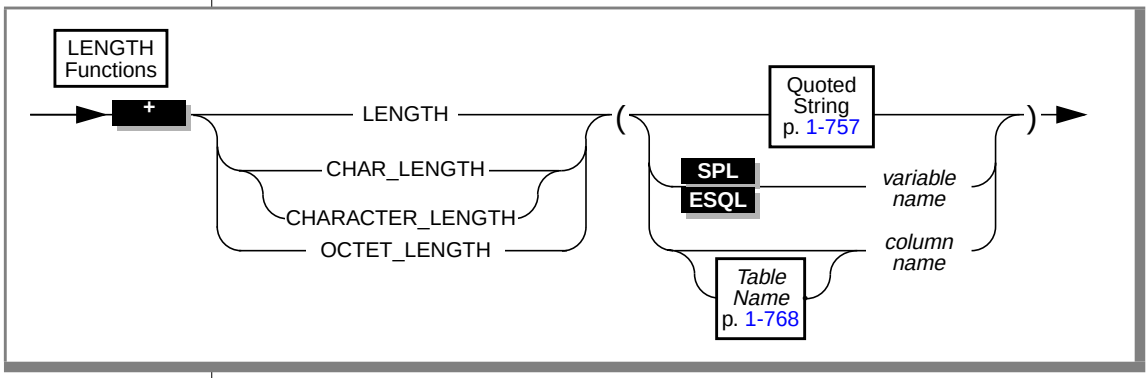
The following example lists the names of all the tables in the current database and their corresponding tblspace number in hexadecimal format. This example is particularly useful because the two most significant bytes in the hexadecimal number constitute the dbspace number. They are used to identify the table in **oncheck** output.

```
SELECT tabname, HEX(partnum) FROM systables
```

The HEX function can operate on an expression, as the following example shows:

```
SELECT HEX(order_num + 1) FROM orders
```

Length Functions



Element	Purpose	Restrictions	Syntax
column name	The name of a column in the specified table.	The column must have a character data type.	Identifier, p. 1-723
variable name	A host variable or procedure variable that contains a character string.	The host variable or procedure variable must have a character data type.	The name of the host variable must conform to language-specific rules for variable names. For the syntax of procedure variables, see "Procedure Call Expressions" on page 1-722.

You can use length functions to determine the length of a column, string, or variable. The length functions are LENGTH, OCTET_LENGTH, and CHAR_LENGTH. Each of these functions has a distinct purpose.

The LENGTH Function

The LENGTH function returns the number of bytes in a character column, not including any trailing spaces. With TEXT or BYTE columns, the LENGTH function returns the full number of bytes in the column, including trailing spaces.

The following example illustrates the use of the LENGTH function:

```
SELECT customer_num, LENGTH(fname) + LENGTH(lname),
    LENGTH('How many bytes is this?')
    FROM customer WHERE LENGTH(company) > 10
```

ESQL

You can use the LENGTH function to return the length of a character variable. ♦

GLS

For information on GLS aspects of the LENGTH function, see Chapter 3 of the *Guide to GLS Functionality*.

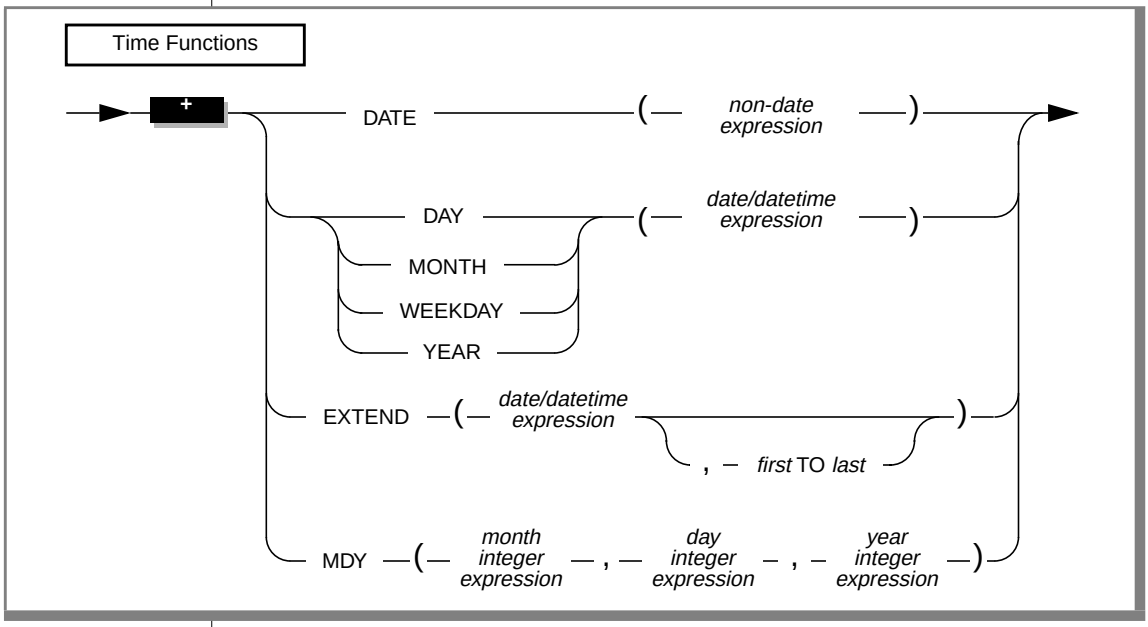
The OCTET_LENGTH Function

The OCTET_LENGTH function returns the number of bytes in a character column, including any trailing spaces. See the *Guide to GLS Functionality* for a discussion of the OCTET_LENGTH function.

The CHAR_LENGTH Function

The CHAR_LENGTH function returns the number of characters (not bytes) in a character column. See the *Guide to GLS Functionality* for a discussion of the CHAR_LENGTH function. ♦

Time Functions



Element	Purpose	Restrictions	Syntax
date/datetime expression	An expression that serves as an argument in the following functions: DAY, MONTH, WEEKDAY, YEAR, and EXTEND	The expression must evaluate to a DATE or DATETIME value.	Expression, p. 1-671
day integer expression	An expression that represents the number of the day of the month	The expression must evaluate to an integer not greater than the number of days in the specified month.	Expression, p. 1-671
first	A qualifier that specifies the first field in the result. If you do not specify *first* and *last* qualifiers, the default value of *first* is YEAR.	The qualifier can be any DATETIME qualifier, as long as it is larger than *last*.	DATETIME Field Qualifier, p. 1-669
last	A qualifier that specifies the last field in the result. If you do not specify *first* and *last* qualifiers, the default value of *last* is FRACTION(3).	The qualifier can be any DATETIME qualifier, as long as it is smaller than *first*.	DATETIME Field Qualifier, p. 1-669

(1 of 2)

Element	Purpose	Restrictions	Syntax
month integer expression	An expression that represents the number of the month	The expression must evaluate to an integer between 1 and 12, inclusive.	Expression, p. 1-671
non-date expression	An expression whose value is to be converted to a DATE data type	You can specify any expression that can be converted to a DATE data type. Usually you specify an expression that evaluates to a CHAR, DATETIME, or INTEGER value.	Expression, p. 1-671
year integer expression	An expression that represents the year	The expression must evaluate to a four-digit integer. You cannot use a two-digit abbreviation.	Expression, p. 1-671

(2 of 2)

DATE Function

The DATE function returns a DATE type value that corresponds to the non-date expression with which you call it. The argument can be any expression that can be converted to a DATE value, usually a CHAR, DATETIME, or INTEGER value. The following WHERE clause specifies a CHAR value for the non-date expression:

```
WHERE order_date < DATE('12/31/93')
```

When the DATE function interprets a CHAR non-date expression, it expects this expression to conform to any DATE format that the **DBDATE** environment specifies. For example, suppose **DBDATE** is set to `Y2MD/` when you execute the following query:

```
SELECT DISTINCT DATE('02/01/1995') FROM ship_info
```

This SELECT statement generates an error because the DATE function cannot convert this non-date expression. The DATE function interprets the first part of the date string (`02`) as the year and the second part (`01`) as the month. For the third part (`1995`), the DATE function encounters four digits when it expects a two-digit day (valid day values must be between `01` and `31`). It therefore cannot convert the value. For the SELECT statement to execute successfully with the `Y2MD/` value for **DBDATE**, the non-date expression would need to be '95/02/01'. For information on the format of **DBDATE**, see Chapter 4 of the *Informix Guide to SQL: Reference*.

When you specify a positive INTEGER value for the non-date expression, the DATE function interprets the value as the number of days after the default date of December 31, 1899. If the integer value is negative, the DATE function interprets the value as the number of days before December 31, 1899. The following WHERE clause specifies an INTEGER value for the non-date expression:

```
WHERE order_date  <  DATE(365)
```

The database server searches for rows with an **order_date** value less than December 31, 1900 (12/31/1899 plus 365 days).

DAY Function

The DAY function returns an integer that represents the day of the month. The following example uses the DAY function with the CURRENT function to compare column values to the current day of the month:

```
WHERE DAY(order_date) > DAY(CURRENT)
```

MONTH Function

The MONTH function returns an integer that corresponds to the month portion of its type DATE or DATETIME argument. The following example returns a number from 1 through 12 to indicate the month when the order was placed:

```
SELECT order_num, MONTH(order_date) FROM orders
```

WEEKDAY Function

The WEEKDAY function returns an integer that represents the day of the week; zero represents Sunday, one represents Monday, and so on. The following lists all the orders that were paid on the same day of the week, which is the current day:

```
SELECT * FROM orders
    WHERE WEEKDAY(paid_date) = WEEKDAY(CURRENT)
```

YEAR Function

The YEAR function returns a four-digit integer that represents the year. The following example lists orders in which the **ship_date** is earlier than the beginning of the current year:

```
SELECT order_num, customer_num FROM orders
    WHERE year(ship_date) < YEAR(TODAY)
```

Similarly, because a DATE value is a simple calendar date, you cannot add or subtract a DATE value with an INTERVAL value whose *last* qualifier is smaller than DAY. In this case, convert the DATE value to a DATETIME value.

EXTEND Function

The EXTEND function adjusts the precision of a DATETIME or DATE value. The expression cannot be a quoted string representation of a DATE value.

If you do not specify *first* and *last* qualifiers, the default qualifiers are YEAR TO FRACTION(3).

If the expression contains fields that are not specified by the qualifiers, the unwanted fields are discarded.

If the *first* qualifier specifies a larger (that is, more significant) field than what exists in the expression, the new fields are filled in with values returned by the CURRENT function. If the *last* qualifier specifies a smaller field (that is, less significant) than what exists in the expression, the new fields are filled in with constant values. A missing MONTH or DAY field is filled in with 1, and the missing HOUR to FRACTION fields are filled in with 0.

In the following example, the first EXTEND call evaluates to the **call_dtime** column value of YEAR TO SECOND. The second statement expands a literal DATETIME so that an interval can be subtracted from it. You must use the EXTEND function with a DATETIME value if you want to add it to or subtract it from an INTERVAL value that does not have all the same qualifiers. The third example updates only a portion of the datetime value, the hour position. The EXTEND function yields just the *hh:mm* part of the datetime. Subtracting 11:00 from the hours / minutes of the datetime yields an INTERVAL value of the difference, plus or minus, and subtracting that from the original value forces the value to 11:00.

```
EXTEND (call_dtime, YEAR TO SECOND)

EXTEND (DATETIME (1989-8-1) YEAR TO DAY, YEAR TO MINUTE)
    - INTERVAL (720) MINUTE (3) TO MINUTE

UPDATE cust_calls SET call_dtime = call_dtime -
(EXTEND(call_dtime, HOUR TO MINUTE) - DATETIME (11:00) HOUR
TO MINUTE) WHERE customer_num = 106
```

MDY Function

The MDY function returns a type DATE value with three expressions that evaluate to integers representing the month, day, and year. The first expression must evaluate to an integer representing the number of the month (1 to 12).

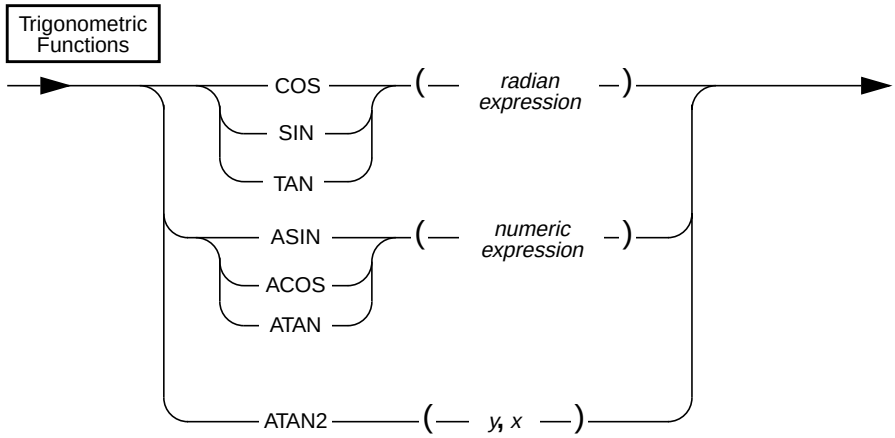
The second expression must evaluate to an integer that represents the number of the day of the month (1 to 28, 29, 30, or 31, as appropriate for the month.)

The third expression must evaluate to a four-digit integer that represents the year. You cannot use a two-digit abbreviation for the third expression. The following example sets the **paid_date** associated with the order number 8052 equal to the first day of the present month:

```
UPDATE orders SET paid_date = MDY(MONTH(TODAY), 1, YEAR(TODAY))
   WHERE po_num = '8052'
```

Trigonometric Functions

A trigonometric function takes an argument, as the following diagram shows.



Element	Purpose	Restrictions	Syntax
numeric expression	A numeric expression that serves as an argument to the ASIN, ACOS, or ATAN functions	The expression must evaluate to a value between -1 and 1, inclusive.	Expression, p. 1-671
radian expression	An expression that evaluates to the number of radians. See "Formulas for Radian Expressions" on page 1-705 for further information on *radian expression*.	The expression must evaluate to a numeric value.	Expression, p. 1-671
x	An expression that represents the *x* coordinate of the rectangular coordinate pair (*x*, *y*)	The expression must evaluate to a numeric value.	Expression, p. 1-671
y	An expression that represents the *y* coordinate of the rectangular coordinate pair (*x*, *y*)	The expression must evaluate to a numeric value.	Expression, p. 1-671

Formulas for Radian Expressions

The COS, SIN, and TAN functions take the number of radians (*radian expression*) as an argument.

If you are using degrees and want to convert degrees to radians, use the following formula:

```
# degrees * p/180= # radians
```

If you are using radians and want to convert radians to degrees, use the following formula:

```
# radians * 180/p = # degrees
```

COS Function

The COS function returns the cosine of a radian expression. The following example returns the cosine of the values of the degrees column in the **anglestbl** table. The expression passed to the COS function in this example converts degrees to radians.

```
SELECT COS(degrees*180/3.1417) FROM anglestbl
```

SIN Function

The SIN function returns the sine of a radian expression. The following example returns the sine of the values in the **radians** column of the **anglestbl** table:

```
SELECT SIN(radians) FROM anglestbl
```

TAN Function

The TAN function returns the tangent of a radian expression. The following example returns the tangent of the values in the **radians** column of the **anglestbl** table:

```
SELECT TAN(radians) FROM anglestbl
```

ACOS Function

The ACOS function returns the arc cosine of a numeric expression. The following example returns the arc cosine of the value (-0.73) in radians:

```
SELECT ACOS(-0.73) FROM anglestbl
```

ASIN Function

The ASIN function returns the arc sine of a numeric expression. The following example returns the arc sine of the value (-0.73) in radians:

```
SELECT ASIN(-0.73) FROM anglestbl
```

ATAN Function

The ATAN function returns the arc tangent of a numeric expression. The following example returns the arc tangent of the value (-0.73) in radians:

```
SELECT ATAN(-0.73) FROM anglestbl
```

ATAN2 Function

The ATAN2 function computes the angular component of the polar coordinates (*r*, θ) associated with (*x*, *y*). The following example compares *angles* to θ for the rectangular coordinates (4, 5):

```
WHERE angles > ATAN2(4,5)      --determines θ for (4,5) and
                                 compares to angles
```

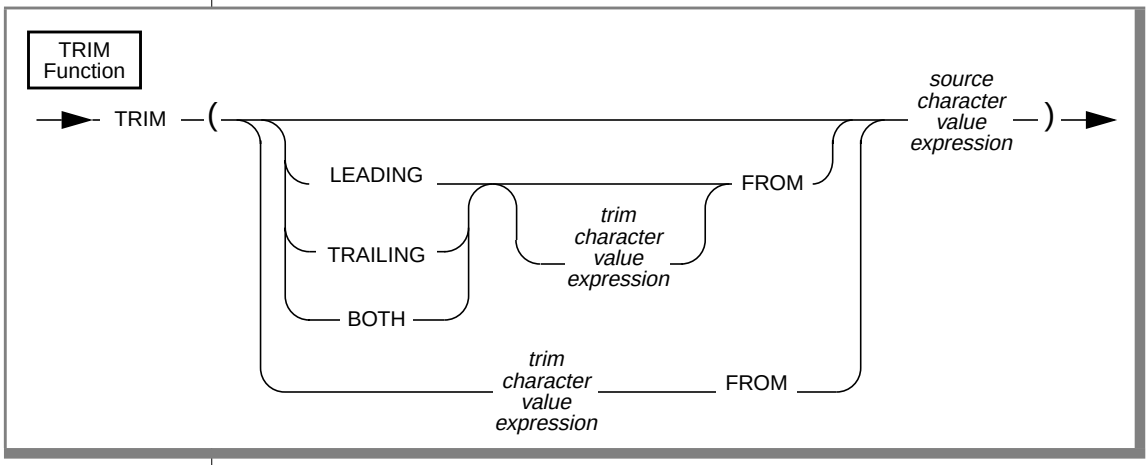
You can determine the length of the radial coordinate *r* using the expression shown in the following example:

```
SQRT(POW(x,2) + POW(y,2))      --determines r for (x,y)
```

You can determine the length of the radial coordinate *r* for the rectangular coordinates (4,5) using the expression shown in the following example:

```
SQRT(POW(4,2) + POW(5,2))      --determines r for (4,5)
```

TRIM Function



Element	Purpose	Restrictions	Syntax
trim character value expression	An expression that evaluates to a single character or null	This expression must be a character expression.	Quoted String, p. 1-757
source character value expression	An arbitrary character string expression, including a column or another TRIM function	This expression cannot be a host variable.	Quoted String, p. 1-757

Use the TRIM function to remove leading or trailing (or both) pad characters from a string. The TRIM function returns a VARCHAR string that is identical to the character string passed to it, except that any leading or trailing pad characters, if specified, are removed. If no trim specification (LEADING, TRAILING, or BOTH) is specified, then BOTH is assumed. If no *trim character value expression* is used, a single space is assumed. If either the *trim character value expression* or the *source character value expression* evaluates to null, the result of the trim function is null. The maximum length of the resultant string must be 255 or less, because the VARCHAR data type supports only 255 characters.

Some generic uses for the TRIM function are shown in the following example:

```
SELECT TRIM (c1) FROM tab;
SELECT TRIM (TRAILING '#' FROM c1) FROM tab;
SELECT TRIM (LEADING FROM c1) FROM tab;
UPDATE c1='xyz' FROM tab WHERE LENGTH(TRIM(c1))=5;
SELECT c1, TRIM(LEADING '#' FROM TRIM(TRAILING '%' FROM
    '###abc%%%')) FROM tab;
```

GLS

When you use the DESCRIBE statement with a SELECT statement that uses the TRIM function in the select list, the described character type of the trimmed column depends on the database server you are using and the data type of the *source character value expression*. See Chapter 7 of the *Guide to GLS Functionality* for further information on the GLS aspects of the TRIM function in ESQL/C. ♦

Fixed Character Columns

The TRIM function can be specified on fixed-length character columns. If the length of the string is not completely filled, the unused characters are padded with blank space. Figure 1-6 shows this concept for the column entry '##A2T##', where the column is defined as CHAR(10).

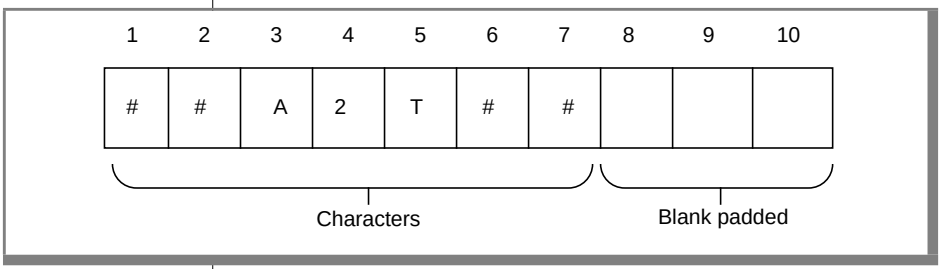


Figure 1-6
Column Entry in a Fixed-Length Character Column

If you want to trim the *trim character value expression* '#' from the column, you need to consider the blank padded spaces as well as the actual characters. For example, if you specify the trim specification BOTH, the result from the trim operation is A2T##, because the TRIM function does not match the blank padded space that follows the string. In this case, the only '#' trimmed are those that precede the other characters. The SELECT statement is shown, followed by Figure 1-7, which presents the result.

```
SELECT TRIM(BOTH '#' FROM col1) FROM taba
```

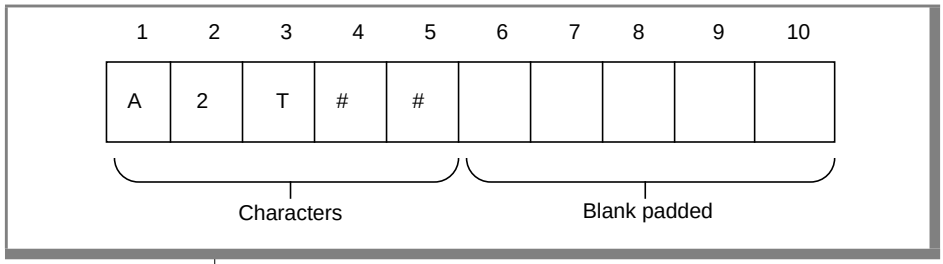


Figure 1-7
Result of TRIM Operation

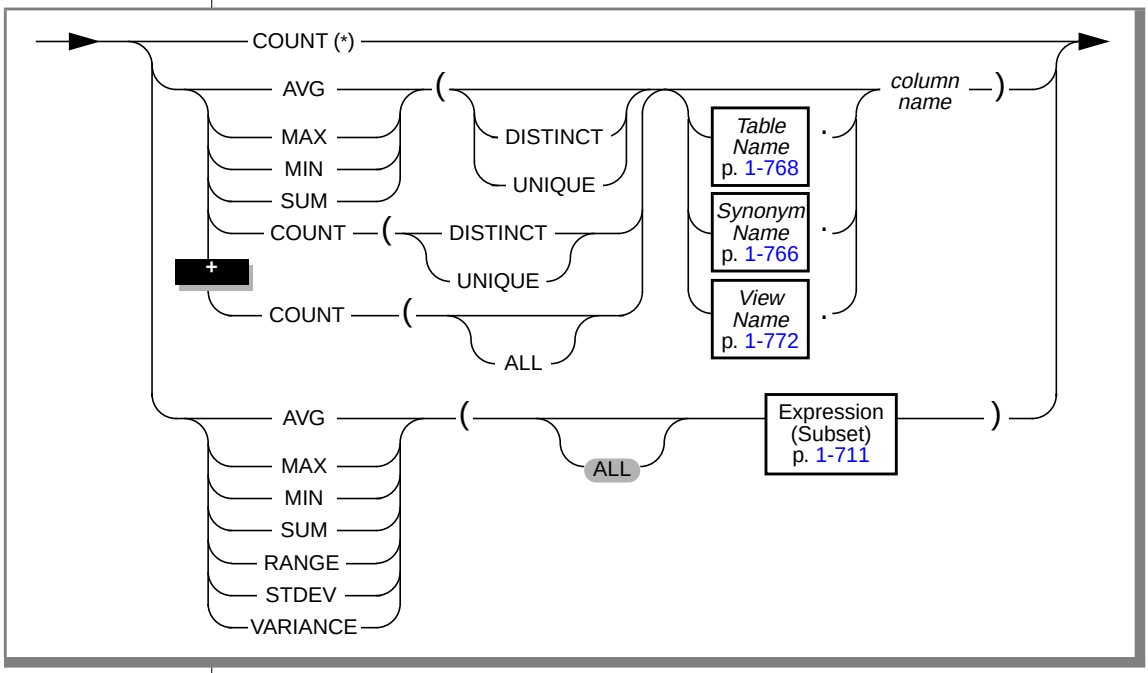
The following SELECT statement removes all occurrences of '#':

```
SELECT TRIM(LEADING '#' FROM TRIM(TRAILING ' ' FROM col1)) FROM taba
```

Aggregate Expressions

An aggregate expression uses an aggregate function to summarize selected database data.

The following diagram shows the syntax of aggregate function expressions.



Element	Purpose	Restrictions	Syntax
column name	The name of the column to which the specified aggregate function is applied	If you specify an aggregate expression and one or more columns in the SELECT clause of a SELECT statement, you must put all the column names that are not used within the aggregate expression or a time expression in the GROUP BY clause. You cannot apply an aggregate function to a BYTE or TEXT column. See "Subset of Expressions Allowed in an Aggregate Expression" on page 1-711 for other general restrictions. For restrictions that depend on the keywords that precede *column name*, see the headings for individual keywords on the following pages.	Identifier, p. 1-723

An aggregate function returns one value for a set of queried rows. The following examples show aggregate functions in SELECT statements:

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013

SELECT COUNT(*) FROM orders WHERE order_num = 1001

SELECT MAX(LENGTH(fname) + LENGTH(lname)) FROM customer
```

If you use an aggregate function and one or more columns in the select list, you must put all the column names that are not used as part of an aggregate or time expression in the GROUP BY clause.

Subset of Expressions Allowed in an Aggregate Expression

The argument of an aggregate function cannot itself contain an aggregate function. You cannot use the aggregate functions found in the following list:

- MAX(AVG(order_num))
- An aggregate function in a WHERE clause unless it is contained in a subquery or if the aggregate is on a correlated column originating from a parent query and the WHERE clause is within a subquery that is within a HAVING clause
- An aggregate function on a BYTE or TEXT column

For the full syntax of expressions, see page 1-671.

Including or Excluding Duplicates in the Row Set

The DISTINCT keyword causes the function to be applied to only unique values from the named column. The UNIQUE keyword is a synonym for the DISTINCT keyword.

The ALL keyword is the opposite of the DISTINCT keyword. If you specify the ALL keyword, all the values that are selected from the named column or expression, including any duplicate values, are used in the calculation.

COUNT(*) Keyword

The COUNT (*) keyword returns the number of rows that satisfy the WHERE clause of a SELECT statement. The following example finds how many rows in the **stock** table have the value HRO in the **manu_code** column:

```
SELECT COUNT(*) FROM stock WHERE manu_code = 'HRO'
```

If the SELECT statement does not have a WHERE clause, the COUNT (*) keyword returns the total number of rows in the table. The following example finds how many rows are in the **stock** table:

```
SELECT COUNT(*) FROM stock
```

If the SELECT statement contains a GROUP BY clause, the COUNT(*) keyword reflects the number of values in each group. The following example is grouped by the first name; the rows are selected if the database server finds more than one occurrence of the same name:

```
SELECT fname, COUNT(*) FROM customer
    GROUP BY fname
    HAVING COUNT(*) > 1
```

If the value of one or more rows is null, the COUNT(*) keyword includes the null columns in the count unless the WHERE clause explicitly omits them.

AVG Keyword

The AVG keyword returns the average of all values in the specified column or expression. You can apply the AVG keyword only to number columns. If you use the DISTINCT keyword, the average (mean) is greater than only the distinct values in the specified column or expression. The query in the following example finds the average price of a helmet:

```
SELECT AVG(unit_price) FROM stock WHERE stock_num = 110
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the AVG keyword returns a null for that column.

MAX Keyword

The MAX keyword returns the largest value in the specified column or expression. Using the DISTINCT keyword does not change the results. The query in the following example finds the most expensive item that is in stock but has not been ordered:

```
SELECT MAX(unit_price) FROM stock
    WHERE NOT EXISTS (SELECT * FROM items
        WHERE stock.stock_num = items.stock_num AND
        stock.manu_code = items.manu_code)
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the MAX keyword returns a null for that column.

MIN Keyword

The MIN keyword returns the lowest value in the column or expression. Using the DISTINCT keyword does not change the results. The following example finds the least expensive item in the **stock** table:

```
SELECT MIN(unit_price) FROM stock
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the MIN keyword returns a null for that column.

SUM Keyword

The SUM keyword returns the sum of all the values in the specified column or expression, as shown in the following example. If you use the DISTINCT keyword, the sum is for only distinct values in the column or expression.

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the SUM keyword returns a null for that column.

You cannot use the SUM keyword with a character column.

COUNT DISTINCT and UNIQUE Keywords

The COUNT DISTINCT keywords return the number of unique values in the column or expression, as the following example shows. If the COUNT function encounters nulls, it ignores them.

```
SELECT COUNT (DISTINCT item_num) FROM items
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the COUNT keyword returns a zero for that column.

The UNIQUE keyword has exactly the same meaning as the DISTINCT keyword when the UNIQUE keyword is used within the COUNT function. The UNIQUE keyword returns the number of unique non-null values in the column or expression.

The following example uses the UNIQUE keyword, but it is equivalent to the preceding example that uses the DISTINCT keyword:

```
SELECT COUNT (UNIQUE item_num) FROM items
```

COUNT column name Option

The COUNT *column name* option returns the total number of non-null values in the column or expression, as the following example shows:

```
SELECT COUNT (item_num) FROM items
```

You can include the ALL keyword before the specified column name for clarity, but the query result is the same whether you include the ALL keyword or omit it.

The following example shows how to include the ALL keyword in the COUNT *column name* option:

```
SELECT COUNT (ALL item_num) FROM items
```

Comparison of the Different Forms of the COUNT Function

You can use the different forms of the COUNT function to retrieve different types of information about a table. The following table summarizes the meaning of each form of the COUNT function.

COUNT Option	Description
COUNT (*)	This option returns the number of rows that satisfy the query. If you do not specify a WHERE clause, this option returns the total number of rows in the table.
COUNT DISTINCT or COUNT UNIQUE	This option returns the number of unique non-null values in the specified column.
COUNT (*column name*) or COUNT (ALL *column name*)	This option returns the total number of non-null values in the specified column.

Some examples can help to show the differences among the different forms of the COUNT function. The following examples pose queries against the **orders** table in the demonstration database. Most of the examples query against the **ship_instruct** column in this table. For information on the structure of the **orders** table and the data in the **ship_instruct** column, see the description of the demonstration database in the *Informix Guide to SQL: Reference*.

Examples of the Count(*) Option

In the following example, the user wants to know the total number of rows in the **orders** table. So the user uses the COUNT(*) function in a SELECT statement without a WHERE clause.

```
SELECT COUNT(*) AS total_rows FROM orders
```

The following table shows the result of this query.

total_rows
23

In the following example, the user wants to know how many rows in the **orders** table have a null value in the **ship_instruct** column. So the user uses the COUNT(*) function in a SELECT statement with a WHERE clause, and specifies the IS NULL condition in the WHERE clause.

```
SELECT COUNT (*) AS no_ship_instruct
    FROM orders
    WHERE ship_instruct IS NULL
```

The following table shows the result of this query.

no_ship_instruct
2

In the following example, the user wants to know how many rows in the **orders** table have the value `express` in the **ship_instruct** column. So the user specifies the COUNT (*) function in the select list and the equals (=) relational operator in the WHERE clause.

```
SELECT COUNT (*) AS ship_express
    FROM ORDERS
    WHERE ship_instruct = 'express'
```

The following table shows the result of this query.

ship_express
6

Examples of the COUNT column name Option

In the following example the user wants to know how many non-null values are in the **ship_instruct** column of the **orders** table. So the user enters the COUNT *column name* function in the select list of the SELECT statement.

```
SELECT COUNT(ship_instruct) AS total_notnulls
    FROM orders
```

The following table shows the result of this query.

total_notnulls
21

The user can also find out how many non-null values are in the **ship_instruct** column by including the ALL keyword in the parentheses that follow the COUNT keyword.

```
SELECT COUNT (ALL ship_instruct) AS all_notnulls
    FROM orders
```

The following table shows that the query result is the same whether you include or omit the ALL keyword.

all_notnulls
21

Examples of the COUNT DISTINCT Option

In the following example, the user wants to know how many unique non-null values are in the **ship_instruct** column of the **orders** table. So the user enters the COUNT DISTINCT function in the select list of the SELECT statement.

```
SELECT COUNT(DISTINCT ship_instruct) AS unique_notnulls
    FROM orders
```

The following table shows the result of this query.

unique_notnulls
16

RANGE Keyword

The RANGE keyword computes the range for a sample of a population. It computes the difference between the maximum and the minimum values, as follows:

```
range(expr) = max(expr) - min(expr)
```

You can apply the RANGE function only to numeric columns. The following query finds the range of ages for a population:

```
SELECT RANGE(age) FROM u_pop
```

As with other aggregates, the RANGE function applies to the rows of a group when the query includes a GROUP BY clause, as shown in the following example:

```
SELECT RANGE(age) FROM u_pop
    GROUP BY birth
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the RANGE function returns a null for that column.

Important: *All computations for the RANGE function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.*

STDEV Keyword

The STDEV keyword computes the standard deviation for a sample of a population. It is the square root of the VARIANCE function.

You can apply the STDEV function only to numeric columns. The following query finds the standard deviation on a population:

```
SELECT STDEV(age) FROM u_pop WHERE u_pop.age > 0
```

As with the other aggregates, the STDEV function applies to the rows of a group when the query includes a GROUP BY clause, as shown in the following example:

```
SELECT STDEV(age) FROM u_pop
    GROUP BY birth
    WHERE STDEV(age) > 0
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the STDEV function returns a null for that column.

Important: *All computations for the STDEV function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.*

VARIANCE Keyword

The VARIANCE keyword returns the variance for a sample of values as an unbiased estimate of the variance of the population. It computes the following value:

```
(SUM(Xi**2) - (SUM(Xi)**2)/N)/(N-1)
```

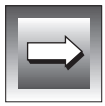
In this example, *Xi* is each value in the column and *N* is the total number of values in the column. You can apply the VARIANCE function only to numeric columns. The following query finds the variance on a population:

```
SELECT VARIANCE(age) FROM u_pop WHERE u_pop.age > 0
```

As with the other aggregates, the VARIANCE function applies to the rows of a group when the query includes a GROUP BY clause, as shown in the following example:

```
SELECT VARIANCE(age) FROM u_pop
    GROUP BY birth
    WHERE VARIANCE(age) > 0
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the VARIANCE function returns a null for that column.



Important: *All computations for the VARIANCE function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.*

Summary of Aggregate Function Behavior

An example can help to summarize the behavior of the aggregate functions. Assume that the **testtable** table has a single INTEGER column that is named **a_number**. The contents of this table are as follows.

a_number
2
2
2
3
3
4
(null)

You can use aggregate functions to obtain different types of information about the **a_number** column and the **testtable** table. In the following example, the user specifies the AVG function to obtain the average of all the non-null values in the **a_number** column:

```
SELECT AVG(a_number) AS average_number
    FROM testtable
```

The following table shows the result of this query.

average_number
2.66666666666667

You can use the other aggregate functions in SELECT statements that are similar to the one shown in the preceding example. If you enter a series of SELECT statements that have different aggregate functions in the select list and do not have a WHERE clause, you receive the results that the following table shows.

Function	Results
COUNT(*)	7
AVG	2.66666666666667
AVG (DISTINCT)	3.00000000000000
MAX	4
MAX(DISTINCT)	4
MIN	2
MIN(DISTINCT)	2
SUM	16
SUM(DISTINCT)	9
COUNT(DISTINCT)	3
COUNT(ALL)	6

(1 of 2)

Function	Results
RANGE	2
STDEV	0.81649658092773
VARIANCE	0.66666666666667

(2 of 2)

Error Checking with Aggregate Functions

ESQL

Aggregate functions always return one row; if no rows are selected, the function returns a null. You can use the COUNT (*) keyword to determine whether any rows were selected, and you can use an indicator variable to determine whether any selected rows were empty. Fetching a row with a cursor associated with an aggregate function always returns one row; hence, 100 for end of data is never returned into the **sqlcode** variable for a first fetch attempt.

You can also use the GET DIAGNOSTICS statement for error checking. See the GET DIAGNOSTICS statement in this manual. ♦

Using Arithmetic Operators with Expressions

You can combine expressions with arithmetic operators to make complex expressions. You cannot combine expressions that use aggregate functions with column expressions. The following examples use arithmetic operators:

```
quantity * total_price

price * 2

COUNT(*) + 2
```

If any value that participates in an arithmetic expression is null, the value of the entire expression is null, as shown in the following example:

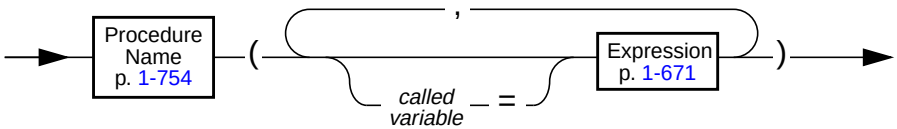
```
SELECT order_num, ship_charge/ship_weight FROM orders
    WHERE order_num = 1023
```

If either **ship_charge** or **ship_weight** is null, the value returned for the expression **ship_charge** / **ship_weight** is also null. If the expression **ship_charge** / **ship_weight** is used in a condition, its truth value is unknown.

If you combine a DATETIME value with one or more INTERVAL values, all the fields of the INTERVAL value must be present in the DATETIME value; no implicit EXTEND function is performed. In addition, you cannot use YEAR to MONTH intervals with DAY to SECOND intervals.

Procedure Call Expressions

The following diagram shows procedure call expressions.



Element	Purpose	Restrictions	Syntax
called variable	The name of a parameter for which you supply an argument to the procedure. The parameter name is originally specified in an CREATE PROCEDURE statement, then used in an EXECUTE PROCEDURE statement.	If you use the *called variable* option for any argument in the called procedure, you must use it for all arguments in the procedure. That is, you must use the *called variable = expression* syntax for all or none of the arguments in the called procedure.	DEFINE statement, p. 2-8

Some typical procedure call expressions are shown in the following examples. The first example omits the *called variable* option, and the second example uses the *called variable* option.

```
read_address('Miller')
read_address(lastname = 'Miller')
```

References

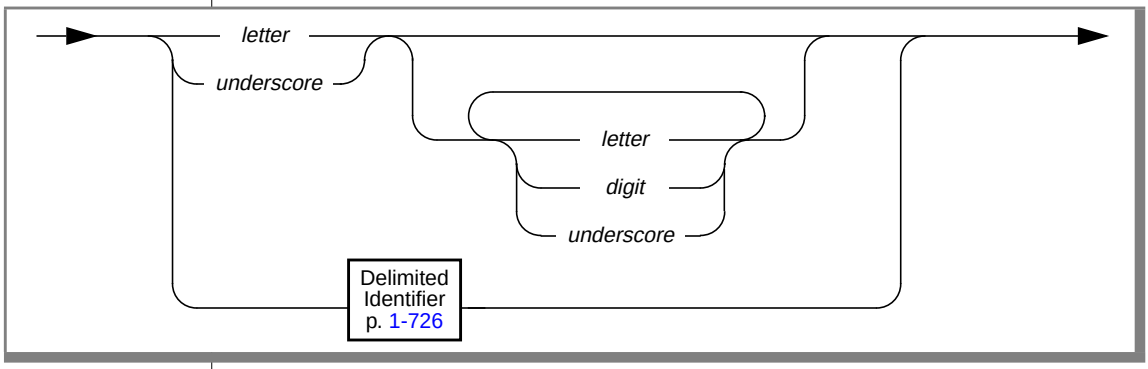
In the *Informix Guide to SQL: Tutorial*, see the discussion of expressions in the SELECT statement in Chapter 2.

In the *Guide to GLS Functionality*, see the discussions of column expressions in Chapter 3, the discussion of length functions in Chapter 3, and the discussion of the TRIM function in Chapter 7.

Identifier

An identifier specifies the simple name of a database object, such as a column, table, index, or view. Use the Identifier segment whenever you see a reference to an identifier in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
digit	An integer that forms part of the identifier	You must specify a number between 0 and 9, inclusive.	Literal Number, p. 1-752
letter	A letter that forms part of the identifier	If you are using the default locale, a *letter* must be an uppercase or lowercase character in the range a to z (in the ASCII code set). If you are using a nondefault locale, *letter* must be an alphabetic character that the locale supports. See "Support for Non-ASCII Characters in Identifiers" on page 1-726 for further information.	Letters are literal values that you enter from the keyboard.
underscore	An underscore character that forms part of the identifier	You cannot substitute a space character, dash, hyphen, or any other nonalphanumeric character for the underscore character.	The underscore character (_) is a literal value that you enter from the keyboard.

Usage

An identifier can contain up to 18 bytes, inclusive.

SE

Database names are limited to 10 bytes. ♦

Use of Reserved Words as Identifiers

Although you can use almost any word as an identifier, syntactic ambiguities can result from using reserved words as identifiers in SQL statements. The statement might fail or might not produce the expected results. See "Potential Ambiguities and Syntax Errors" on page 1-729 for a discussion of the syntactic ambiguities that can result from using reserved words as identifiers and an explanation of workarounds for these problems.

Delimited identifiers provide the easiest and safest way to use a reserved word as an identifier without causing syntactic ambiguities. No workarounds are necessary when you use a reserved word as a delimited identifier. See "Delimited Identifiers" on page 1-726 for the syntax and usage of delimited identifiers.

Tip: *If you receive an error message that seems unrelated to the statement that caused the error, check to determine whether the statement uses a reserved word as an undelimited identifier.*

ANSI-Reserved Words

The following list specifies all the ANSI-reserved words (that is, reserved words in the ANSI SQL standard).

ADA	execute	order
all	exists	pascal
and	fetch	pli
any	float	precision
as	for	primary
asc	fortran	procedure
authorization	found	privileges
avg	from	public
begin	go	real
between	goto	rollback

(1 of 2)

by	group	schema
char	having	section
character	in	select
check	indicator	set
close	insert	smallint
cobol	int	some
commit	integer	sql
continue	into	sqlcode
count	is	sqlerror
create	language	sum
current	like	table
cursor	max	to
dec	min	union
decimal	module	unique
declare	not	update
delete	null	user
desc	numeric	values
distinct	of	view
double	on	whenever
end	open	where
escape	option	with
exec	or	work

(2 of 2)

You can flag identifiers as ANSI-reserved words by taking the following steps:

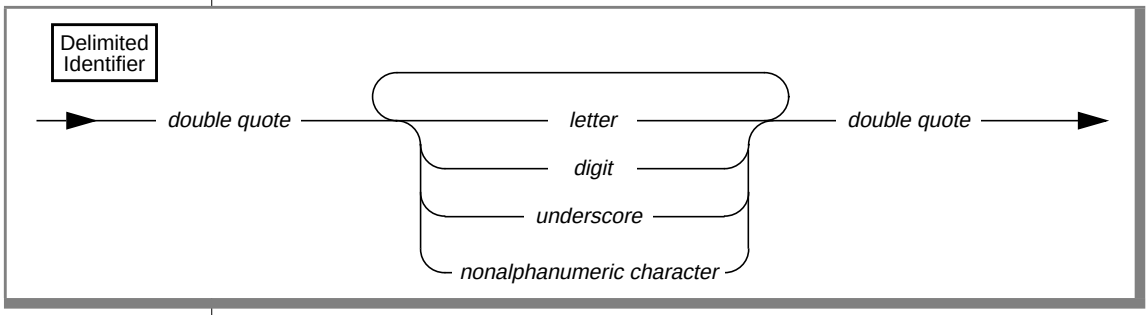
- Set the **DBANSIWARN** environment variable or use the **-ansi** flag at compile time to receive compile-time warnings.
- Set the **DBANSIWARN** environment variable at runtime to receive warning flags set in the **SQLWARN** array of **sqlca**.

Support for Non-ASCII Characters in Identifiers

GLS

If you are using a nondefault locale, you can use any alphabetic character that your locale recognizes as a *letter* in an SQL identifier name. You can use a non-ASCII character as a letter as long as your locale supports it. This feature enables you to use non-ASCII characters in the names of database objects such as indexes, tables, and views. For a list of SQL identifiers that support non-ASCII characters, see Chapter 3 of the *Guide to GLS Functionality*. ♦

Delimited Identifiers



Element	Purpose	Restrictions	Syntax
digit	An integer that forms part of the delimited identifier	You must specify a number between 0 and 9, inclusive.	Literal Number, p. 1-752
double quote	The double-quote character that marks a string as a delimited identifier	If the **DELIMIDENT** environment variable is not set, values within double quotes are treated as quoted strings by the database server.	The double quote character (") is a literal value that you enter from the keyboard.
letter	A letter that forms part of the delimited identifier	Letters in delimited identifiers are case-sensitive. If you are using the default locale, a *letter* must be an uppercase or lowercase character in the range a-z (in the ASCII code set). If you are using a nondefault locale, *letter* must be an alphabetic character that the locale supports. See "Support for Non-ASCII Characters in Delimited Identifiers" on page 1-728 for further information.	Letters are literal values that you enter from the keyboard.
nonalpha-numeric character	A nonalphanumeric character, such as # or $ or space, that forms part of the delimited identifier	If you are using the ASCII code set, you can specify any ASCII nonalphanumeric character.	Nonalphanumeric characters are literal values that you enter from the keyboard.
underscore	An underscore (_) that forms part of the delimited identifier	You can use a dash, hyphen, or any other appropriate character in place of the underscore character.	The underscore (_) is a literal value that you enter from the keyboard.

Delimited identifiers allow you to specify names for database objects that are otherwise identical to SQL reserved keywords, such as TABLE, WHERE, DECLARE, and so on. The only database object for which you cannot use delimited identifiers is database name.

Delimited identifiers are case sensitive.

Delimited identifiers are compliant with the ANSI standard.

Support for Nonalphanumeric Characters

You can use delimited identifiers to specify nonalphanumeric characters in the names of database objects. However, you cannot use delimited identifiers to specify nonalpha characters in the names of storage objects such as dbspaces and blobspaces.

Support for Non-ASCII Characters in Delimited Identifiers

GLS

When you are using a nondefault locale whose code set supports non-ASCII characters, you can specify non-ASCII characters in most delimited identifiers. The rule is that if you can specify non-ASCII characters in the undelimited form of the identifier, you can also specify non-ASCII characters in the delimited form of the same identifier. See Chapter 3 of the *Guide to GLS Functionality* for a list of identifiers that support non-ASCII characters and for information on non-ASCII characters in delimited identifiers. ♦

Effect of DELIMIDENT Environment Variable

To use delimited identifiers, you must set the **DELIMIDENT** environment variable. When you set the **DELIMIDENT** environment variable, database objects in double quotes (") are treated as identifiers and database objects in single quotes (') are treated as strings. If the **DELIMIDENT** environment variable is not set, values within double quotes are also treated as strings.

If the **DELIMIDENT** variable is set, the SELECT statement in the following example must be in single quotes in order to be treated as a quoted string:

```
PREPARE ... FROM 'SELECT * FROM customer'
```

Examples of Delimited Identifiers

The following example shows how to create a table with a case-sensitive table name:

```
CREATE TABLE "Power_Ranger" (...)
```

The following example shows how to create a table whose name includes a space character. If the table name were not in double quotes ("), you could not use a space character or any other nonalpha character except an underscore (_) in the name.

```
CREATE TABLE "My Customers" (...)
```

The following example shows how to create a table that uses a keyword as the table name:

```
CREATE TABLE "TABLE" (...)
```

Using Double Quotes Within a Delimited Identifier

If you want to include a double-quote (") within a delimited identifier, you must precede the double-quote (") with another double-quote (), as shown in the following example:

```
CREATE TABLE "My""Good""Data" (...)
```

Potential Ambiguities and Syntax Errors

Although you can use almost any word as an SQL identifier, syntactic ambiguities can occur. An ambiguous statement might not produce the desired results. The following sections outline some potential pitfalls and workarounds.

Using Functions as Column Names

The following two examples show a workaround for using a function as a column name in a SELECT statement. This workaround applies to the aggregate functions (AVG, COUNT, MAX, MIN, SUM) as well as the function expressions (algebraic, exponential and logarithmic, time, hex, length, dbinfo, trigonometric, and trim functions).

Using **avg** as a column name causes the following example to fail because the database server interprets **avg** as an aggregate function rather than as a column name:

```
SELECT avg FROM mytab -- fails
```

If the **DELIMIDENT** environment variable is set, you could use **avg** as a column name as shown in the following example:

```
SELECT "avg" from mytab -- successful
```

The workaround in following example removes ambiguity by including a table name with the column name:

```
SELECT mytab.avg FROM mytab
```

If you use the keyword TODAY, CURRENT, or USER as a column name, ambiguity can occur, as shown in the following example:

```
CREATE TABLE mytab (user char(10),
    CURRENT DATETIME HOUR TO SECOND,TODAY DATE)

INSERT INTO mytab VALUES('josh','11:30:30','1/22/89')

SELECT user,current,today FROM mytab
```

The database server interprets **user**, **current**, and **today** in the SELECT statement as the SQL functions USER, CURRENT, and TODAY. Thus, instead of returning `josh, 11:30:30,1/22/89`, the SELECT statement returns the current user name, the current time, and the current date.

If you want to select the actual columns of the table, you must write the SELECT statement in one of the following ways:

```
SELECT mytab.user, mytab.current, mytab.today FROM mytab;

EXEC SQL select * from mytab;
```

Using Keywords as Column Names

Specific workarounds exist for using a keyword as a column name in a SELECT statement or other SQL statement. In some cases, there might be more than one suitable workaround.

Using ALL, DISTINCT, or UNIQUE as a Column Name

If you want to use the ALL, DISTINCT, or UNIQUE keywords as column names in a SELECT statement, you can take advantage of a workaround.

First, consider what happens when you try to use one of these keywords without a workaround. In the following example, using **all** as a column name causes the SELECT statement to fail because the database server interprets **all** as a keyword rather than as a column name:

```
SELECT all FROM mytab -- fails
```

You need to use a workaround to make this SELECT statement execute successfully. If the **DELIMIDENT** environment variable is set, you can use **all** as a column name by enclosing **all** in double quotes. In the following example, the SELECT statement executes successfully because the database server interprets **all** as a column name:

```
SELECT "all" from mytab -- successful
```

The workaround in the following example uses the keyword ALL with the column name **all**:

```
SELECT ALL all FROM mytab
```

The rest of the examples in this section show workarounds for using the keywords UNIQUE or DISTINCT as a column name in a CREATE TABLE statement.

Using **unique** as a column name causes the following example to fail because the database server interprets **unique** as a keyword rather than as a column name:

```
CREATE TABLE mytab (unique INTEGER) -- fails
```

The workaround shown in the following example uses two SQL statements. The first statement creates the column **mycol**; the second renames the column **mycol** to **unique**.

```
CREATE TABLE mytab (mycol INTEGER)

RENAME COLUMN mytab.mycol TO unique
```

The workaround in the following example also uses two SQL statements. The first statement creates the column **mycol**; the second alters the table, adds the column **unique**, and drops the column **mycol**.

```
CREATE TABLE mytab (mycol INTEGER)

ALTER TABLE mytab
    ADD (unique integer)
    DROP (mycol)
```

Using INTERVAL or DATETIME as a Column Name

The examples in this section show workarounds for using the keyword INTERVAL (or DATETIME) as a column name in a SELECT statement.

Using **interval** as a column name causes the following example to fail because the database server interprets **interval** as a keyword and expects it to be followed by an INTERVAL qualifier:

```
SELECT interval FROM mytab -- fails
```

If the **DELIMIDENT** environment variable is set, you could use **interval** as a column name, as shown in the following example:

```
SELECT "interval" from mytab -- successful
```

The workaround in the following example removes ambiguity by specifying a table name with the column name:

```
SELECT mytab.interval FROM mytab;
```

The workaround in the following example includes an owner name with the table name:

```
SELECT josh.mytab.interval FROM josh.mytab;
```

Using rowid as a Column Name

Every nonfragmented table has a virtual column named **rowid**. To avoid ambiguity, you cannot use **rowid** as a column name. Performing the following actions causes an error:

- Creating a table or view with a column named **rowid**
- Altering a table by adding a column named **rowid**
- Renaming a column to **rowid**

You can, however, use the term **rowid** as a table name.

```
CREATE TABLE rowid (column INTEGER,
    date DATE, char CHAR(20))
```

Important: *Informix recommends that you use primary keys as an access method rather than exploiting the rowid column.*

Using Keywords as Table Names

The examples in this section show workarounds that involve owner naming when you use the keyword STATISTICS or OUTER as a table name. This workaround also applies to the use of STATISTICS or OUTER as a view name or synonym.

Using **statistics** as a table name causes the following example to fail because the database server interprets it as part of the UPDATE STATISTICS syntax rather than as a table name in an UPDATE statement:

```
UPDATE statistics SET mycol = 10
```

The workaround in the following example specifies an owner name with the table name, to avoid ambiguity:

```
UPDATE josh.statistics SET mycol = 10
```

Using **outer** as a table name causes the following example to fail because the database server interprets **outer** as a keyword for performing an outer join:

```
SELECT mycol FROM outer -- fails
```

The workaround in the following example uses owner naming to avoid ambiguity:

```
SELECT mycol FROM josh.outer
```

Workarounds That Use the Keyword AS

In some cases, although a statement is not ambiguous and the syntax is correct, the database server returns a syntax error. The preceding pages show existing syntactic workarounds for several situations. You can use the AS keyword to provide a workaround for the exceptions.

You can use the AS keyword in front of column labels or table aliases.

The following example uses the AS keyword with a column label:

```
SELECT column-name AS display-label FROM table-name
```

The following example uses the AS keyword with a table alias:

```
SELECT select-list FROM table-name AS table-alias
```

Using AS with Column Labels

The examples in this section show workarounds that use the AS keyword with a column label. The first two examples show how you can use the keyword UNITS (or YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION) as a column label.

Using **units** as a column label causes the following example to fail because the database server interprets it as a DATETIME qualifier for the column named **mycol**:

```
SELECT mycol units FROM mytab
```

The workaround in the following example includes the AS keyword:

```
SELECT mycol AS units FROM mytab;
```

The following examples show how the AS or FROM keyword can be used as a column label.

Using **as** as a column label causes the following example to fail because the database server interprets **as** as identifying **from** as a column label and thus finds no required FROM clause:

```
SELECT mycol as from mytab -- fails
```

The following example repeats the AS keyword:

```
SELECT mycol AS as from mytab
```

Using **from** as a column label causes the following example to fail because the database server expects a table name to follow the first **from**:

```
SELECT mycol from FROM mytab -- fails
```

The following example uses the AS keyword to identify the first **from** as a column label:

```
SELECT mycol AS from FROM mytab
```

Using AS with Table Aliases

The examples in this section show workarounds that use the AS keyword with a table alias. The first pair shows how to use the ORDER, FOR, GROUP, HAVING, INTO, UNION, WITH, CREATE, GRANT, or WHERE keyword as a table alias.

Using **order** as a table alias causes the following example to fail because the database server interprets **order** as part of an ORDER BY clause:

```
SELECT * FROM mytab order -- fails
```

The workaround in the following example uses the keyword AS to identify **order** as a table alias:

```
SELECT * FROM mytab AS order;
```

The following two examples show how to use the keyword WITH as a table alias.

Using **with** as a table alias causes the following example to fail because the database server interprets the keyword as part of the WITH CHECK OPTION syntax:

```
EXEC SQL select * from mytab with; -- fails
```

The workaround in the following example uses the keyword AS to identify **with** as a table alias:

```
EXEC SQL select * from mytab as with;
```

The following two examples show how to use the keyword CREATE (or GRANT) as a table alias.

Using **create** as a table alias causes the following example to fail because the database server interprets the keyword as part of the syntax to create an entity such as a table, synonym, or view:

```
EXEC SQL select * from mytab create; -- fails
```

The workaround in the following example uses the keyword AS to identify **create** as a table alias:

```
EXEC SQL select * from mytab as create;
```

Fetching Keywords as Cursor Names

In a few situations, no workaround exists for the syntactic ambiguity that occurs when a keyword is used as an identifier in an SQL program.

In the following example, the FETCH statement specifies a cursor named **next**. The FETCH statement generates a syntax error because the preprocessor interprets **next** as a keyword, signifying the next row in the active set and expects a cursor name to follow **next**. This occurs whenever the keyword NEXT, PREVIOUS, PRIOR, FIRST, LAST, CURRENT, RELATIVE, or ABSOLUTE is used as a cursor name.

```
/* This code fragment fails */
EXEC SQL declare next cursor for
    select customer_num, lname from customer;

EXEC SQL open next;
EXEC SQL fetch next into :cnum, :lname;
```

Using Keywords as Procedure Variable Names

If you use any of the following keywords as identifiers for variables in a procedure, you can create ambiguous syntax.

CURRENT	OFF
DATETIME	ON
GLOBAL	PROCEDURE
INTERVAL	SELECT
NULL	

Using CURRENT, DATETIME, INTERVAL, and NULL in INSERT

You cannot use the CURRENT, DATETIME, INTERVAL, or NULL keyword as the name of a procedure with the INSERT statement.

For example, if you define a variable called **null**, when you try to insert the value **null** into a column, you receive a syntax error, as shown in the following example:

```
CREATE PROCEDURE problem()
.
.
.
DEFINE null INT;
LET null = 3;
INSERT INTO tab VALUES (null); -- error, inserts NULL, not 3
```

Using NULL and SELECT in a Condition

If you define a variable with the name *null* or *select*, using it in a condition that uses the IN keyword is ambiguous. The following example shows three conditions that cause problems: in an IF statement, in a WHERE clause of a SELECT statement, and in a WHILE condition:

```
CREATE PROCEDURE problem()
.
.
.
DEFINE x,y,select, null, INT;
DEFINE pfname CHAR[15];
LET x = 3; LET select = 300;
LET null = 1;
IF x IN (select, 10, 12) THEN LET y = 1; -- problem if

IF x IN (1, 2, 4) THEN
SELECT customer_num, fname INTO y, pfname FROM customer
    WHERE customer IN (select , 301 , 302, 303); -- problem in

WHILE x IN (null, 2) -- problem while
.
.
.
END WHILE;
```

You can use the variable *select* in an IN list if you ensure it is not the first element in the list. The workaround in the following example corrects the IF statement shown in the preceding example:

```
IF x IN (10, select, 12) THEN LET y = 1; -- problem if
```

No workaround exists to using *null* as a variable name and attempting to use it in an IN condition.

Using ON, OFF, or PROCEDURE with TRACE

If you define a procedure variable called *on*, *off*, or *procedure*, and you attempt to use it in a TRACE statement, the value of the variable does not trace. Instead, the TRACE ON, TRACE OFF, or TRACE PROCEDURE statements execute. You can trace the value of the variable by making the variable into a more complex expression. The following example shows the ambiguous syntax and the workaround:

```
DEFINE on, off, procedure INT;

TRACE on;       --ambiguous
TRACE 0+ on;    --ok
TRACE off;      --ambiguous
TRACE ''||off;  --ok

TRACE procedure;--ambiguous
TRACE 0+procedure;--ok
```

Using GLOBAL as a Variable Name

If you attempt to define a variable with the name *global*, the define operation fails. The syntax shown in the following example conflicts with the syntax for defining global variables:

```
DEFINE global INT; -- fails;
```

If the **DELIMIDENT** environment variable is set, you could use **global** as a variable name, as shown in the following example:

```
DEFINE "global" INT; -- successful
```

Using EXECUTE, SELECT, or WITH as Cursor Names

Do not use an EXECUTE, SELECT, or WITH keyword as the name of a cursor. If you try to use one of these keywords as the name of a cursor in a FOREACH statement, the cursor name is interpreted as a keyword in the FOREACH statement. No workaround exists.

The following example does not work:

```
DEFINE execute INT;
FOREACH execute FOR SELECT col1 -- error, looks like
                                -- FOREACH EXECUTE PROCEDURE
     INTO var1 FROM tab1; --
```

SELECT Statements in WHILE and FOR Statements

If you use a SELECT statement in a WHILE or FOR loop, and if you need to enclose it in parentheses, enclose the entire SELECT statement in a BEGIN...END block. The SELECT statement in the first WHILE statement in the following example is interpreted as a call to the procedure **var1**; the second WHILE statement is interpreted correctly:

```
DEFINE var1, var2 INT;
WHILE var2 = var1
    SELECT col1 INTO var3 FROM TAB -- error, seen as call var1()
    UNION
    SELECT co2 FROM tab2;
END WHILE

WHILE var2 = var1
    BEGIN
        SELECT col1 INTO var3 FROM TAB -- ok syntax
        UNION
        SELECT co2 FROM tab2;
    END
END WHILE
```

The SET Keyword in the ON EXCEPTION Statement

If you use a statement that begins with the keyword SET inside the statement ON EXCEPTION, you must enclose it in a BEGIN...END block. The following list shows some of the SQL statements that begin with the keyword SET.

SET	SET LOCK MODE
SET DEBUG FILE	SET LOG
SET EXPLAIN	SET OPTIMIZATION
SET ISOLATION	SET PDQPRIORITY

The following examples show incorrect and correct use of a SET LOCK MODE statement inside an ON EXCEPTION statement.

The following ON EXCEPTION statement returns an error because the SET LOCK MODE statement is not enclosed in a BEGIN...END block:

```
ON EXCEPTION IN (-107)
    SET LOCK MODE TO WAIT; -- error, value expected, not 'lock'
END EXCEPTION
```

The following ON EXCEPTION statement executes successfully because the SET LOCK MODE statement is enclosed in a BEGIN...END block:

```
ON EXCEPTION IN (-107)
    BEGIN
    SET LOCK MODE TO WAIT; -- ok
    END
END EXCEPTION
```

References

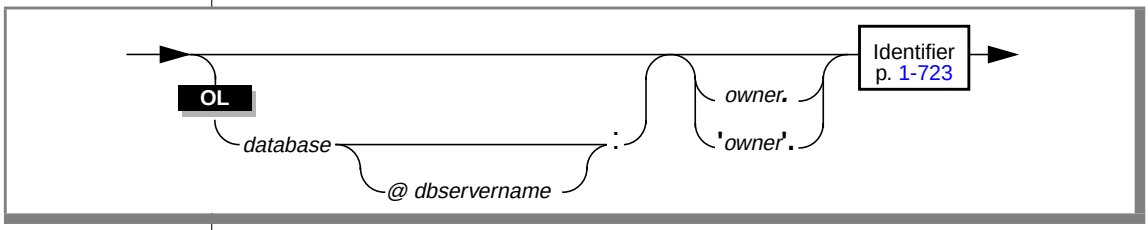
In the *INFORMIX-OnLine Dynamic Server Performance Guide*, see the owner-naming discussion.

In Chapter 3 of the *Guide to GLS Functionality*, see the discussion of identifiers that support non-ASCII characters and the discussion of non-ASCII characters in delimited identifiers.

Index Name

The Index Name segment specifies the name of an index. Use the Index Name segment whenever you see a reference to an index name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the index resides	The database must exist.	Database Name, p. 1-660
dbservername	The name of the OnLine database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-660
owner	The user name of the owner of the index	If you are using an ANSI-compliant database, you must specify the owner for an index that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The actual name of the index is an SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of indexes. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

If you are creating an index, the *name* must be unique within a database.

ANSI

The *owner.name* combination is case sensitive. In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the table owner is stored as uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-770. ♦

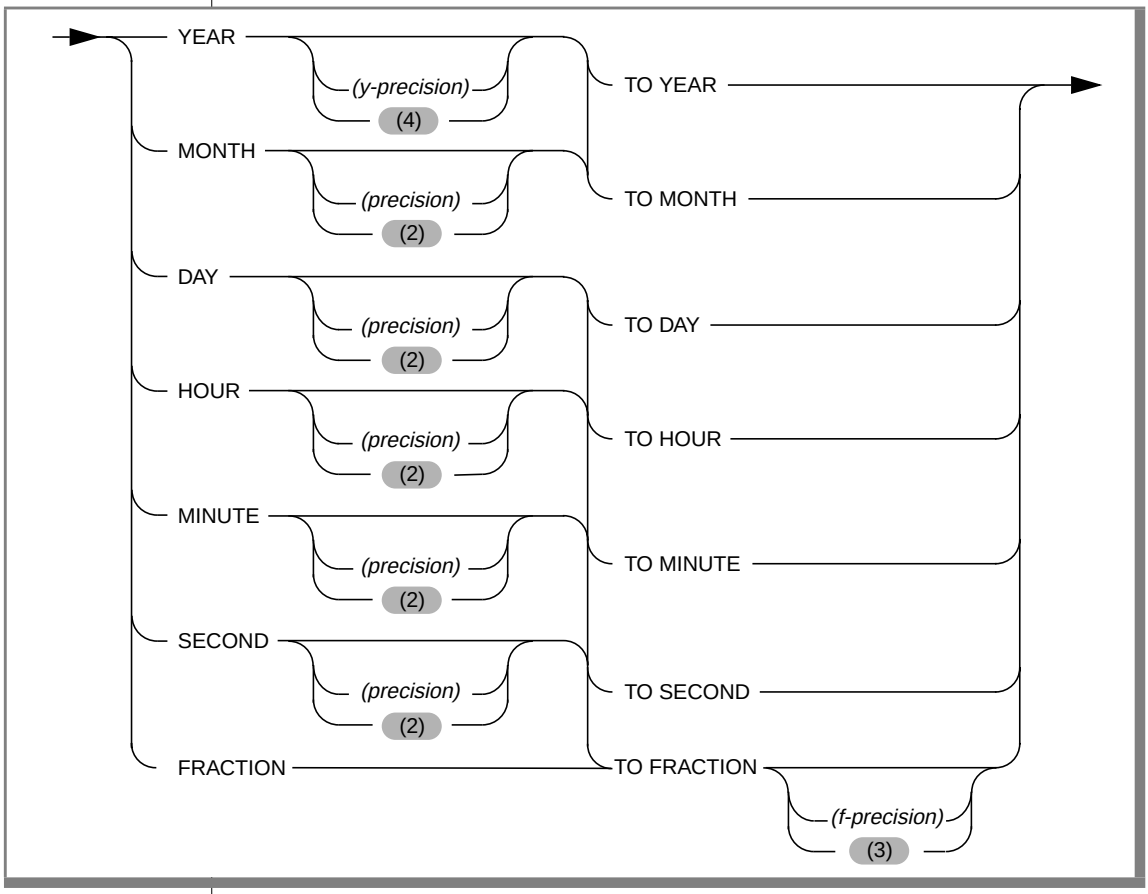
References

See the CREATE INDEX statement in this manual for information on defining indexes.

INTERVAL Field Qualifier

The INTERVAL field qualifier specifies the units for an INTERVAL value. Use the INTERVAL Field Qualifier segment whenever you see a reference to an INTERVAL field qualifier in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
f-precision	The maximum number of digits used in the fraction field. The default value of *f-precision* is 3.	The maximum value that you can specify in *f-precision* is 5.	Literal Number, p. 1-752
precision	The number of digits in the largest number of months, days, hours, or minutes that the interval can hold. The default value of *precision* is 2.	The maximum value that you can specify in *precision* is 9.	Literal Number, p. 1-752
y-precision	The number of digits in the largest number of years that the interval can hold. The default value of *y-precision* is 4.	The maximum value that you can specify in *y-precision* is 9.	Literal Number, p. 1-752

Usage

The next two examples show INTERVAL data types of the YEAR TO MONTH type. The first example can hold an interval of up to 999 years and 11 months, because it gives 3 as the precision of the year field. The second example uses the default precision on the year field, so it can hold an interval of up to 9,999 years and 11 months.

```
YEAR (3) TO MONTH

YEAR TO MONTH
```

When you want a value to contain only one field, the first and last qualifiers are the same. For example, an interval of whole years is qualified as YEAR TO YEAR or YEAR (5) TO YEAR, for an interval of up to 99,999 years.

The following examples show several forms of INTERVAL qualifiers:

```
YEAR(5) TO MONTH

DAY (5) TO FRACTION(2)

DAY TO DAY

FRACTION TO FRACTION (4)
```

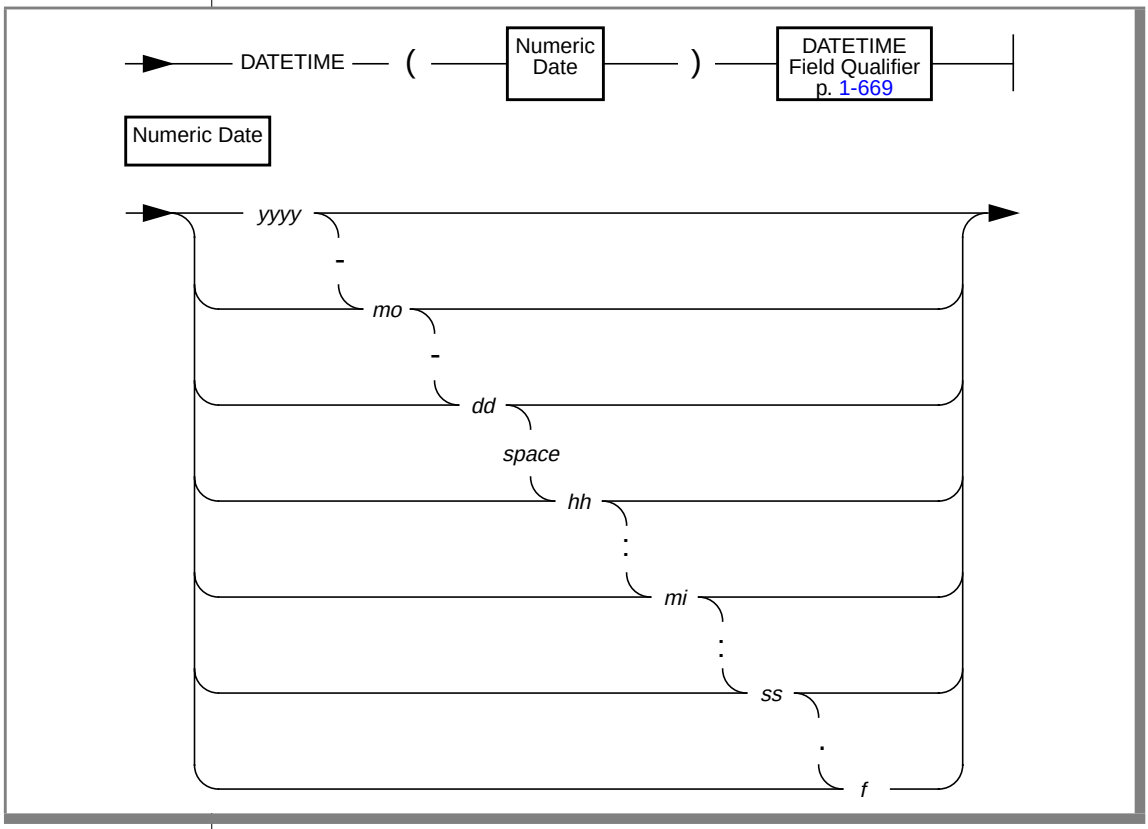
References

In the *Informix Guide to SQL: Reference*, see the INTERVAL data type in Chapter 3 for information about specifying INTERVAL field qualifiers and using INTERVAL data in arithmetic and relational operations.

Literal DATETIME

The literal DATETIME segment specifies a literal DATETIME value. Use the literal DATETIME segment whenever you see a reference to a literal DATETIME in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dd	The day expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-752
f	The decimal fraction of a second expressed in digits	You can specify up to 5 digits.	Literal Number, p. 1-752
hh	The hour expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-752
mi	The minute expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-752
mo	The month expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-752
space	A space character	You cannot specify more than 1 space character.	The space character is a literal value that you enter by pressing the space bar on the keyboard.
ss	The second expressed in digits	You can specify up to 2 digits.	Literal Number, p. 1-752
yyyy	The year expressed in digits	You can specify up to 4 digits. If you specify 2 digits, the database server uses the setting of the **DBCENTURY** environment variable to extend the year value. If the **DBCENTURY** environment variable is not set, the database server uses the current century to extend the year value.	Literal Number, p. 1-752

Usage

You must specify both a numeric date and a DATETIME field qualifier for this date in the Literal DATETIME segment. The DATETIME field qualifier must correspond to the numeric date you specify. For example, if you specify a numeric date that includes a year as the largest unit and a minute as the smallest unit, you must specify YEAR TO MINUTE as the DATETIME field qualifier.

The following examples show literal DATETIME values:

```
DATETIME (93-3-6) YEAR TO DAY

DATETIME (09:55:30.825) HOUR TO FRACTION

DATETIME (93-5) YEAR TO MONTH
```

The following example shows a literal DATETIME value used with the EXTEND function:

```
EXTEND (DATETIME (1993-8-1) YEAR TO DAY, YEAR TO MINUTE)
    - INTERVAL (720) MINUTE (3) TO MINUTE
```

References

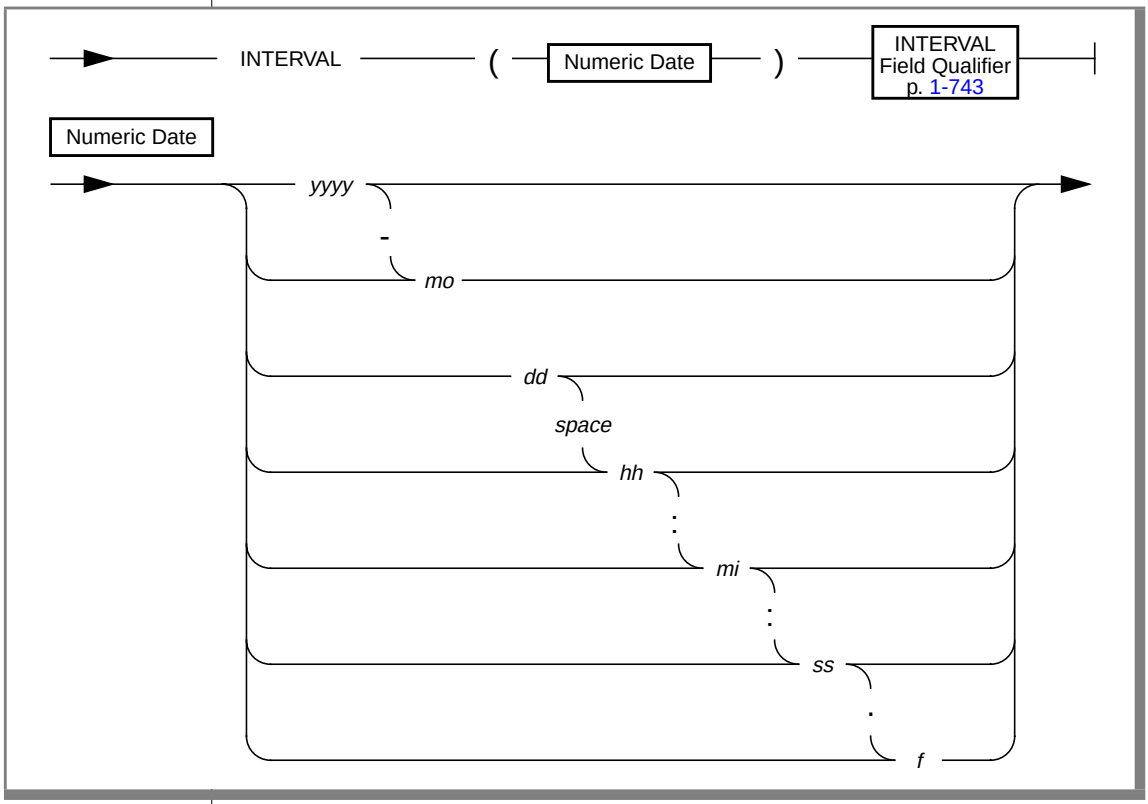
In the *Informix Guide to SQL: Reference*, see the DATETIME data type in Chapter 3 and the **DBCENTURY** environment variable in Chapter 4.

In Chapter 1 of the *Guide to GLS Functionality*, see the discussion of customizing DATETIME values for a locale.

Literal INTERVAL

The Literal INTERVAL segment specifies a literal INTERVAL value. Use the Literal INTERVAL segment whenever you see a reference to a literal INTERVAL in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
dd	The number of days	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-752
f	The decimal fraction of a second	You can specify up to 5 digits, depending on the precision given to the fractional portion in the INTERVAL field qualifier.	Literal Number, p. 1-752
hh	The number of hours	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-752
mi	The number of minutes	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-752
mo	The number of months	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-752
space	A space character	You cannot use any other character in place of the space character.	The space character is a literal value that you enter by pressing the space bar on the keyboard.
ss	The number of seconds	The maximum number of digits allowed is 2, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-752
yyyy	The number of years	The maximum number of digits allowed is 4, unless this is the first field and the precision is specified differently by the INTERVAL field qualifier.	Literal Number, p. 1-752

Usage

The following examples show literal INTERVAL values:

```
INTERVAL (3-6) YEAR TO MONTH
INTERVAL (09:55:30.825) HOUR TO FRACTION
INTERVAL (40 5) DAY TO HOUR
```

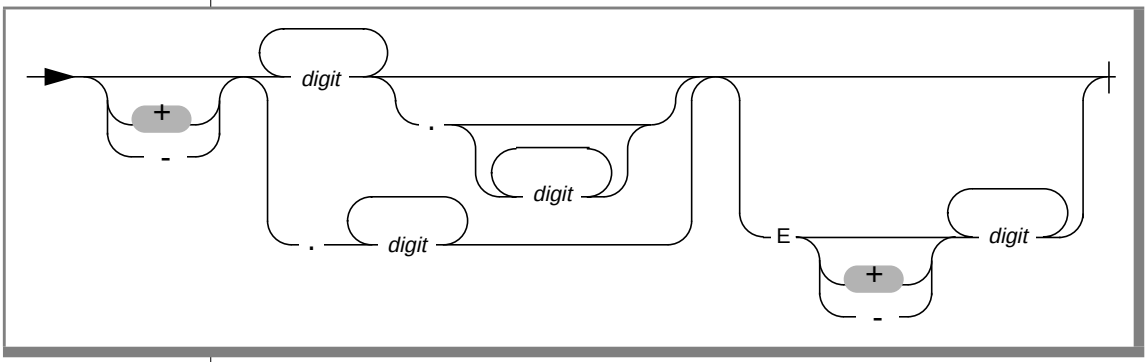
References

In the *Informix Guide to SQL: Reference*, see the INTERVAL data type in Chapter 3 for information about using INTERVAL data in arithmetic and relational operations.

Literal Number

A literal number is an integer or noninteger (floating) constant. Use the Literal Number segment whenever you see a reference to a literal number in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
digit	A digit that forms part of the literal number. See "Floating and Decimal Numbers" on page 1-753 for the significance of digits that follow the decimal point or the E symbol.	You must specify a value between 0 and 9, inclusive.	Digits are literal values that you enter from the keyboard.

Usage

Literal numbers do not contain embedded commas; you cannot use a comma to indicate a decimal point. You can precede literal numbers with a plus or a minus sign.

Integers

Integers do not contain decimal points. The following examples show some integers:

```
10          -27          25567
```

Floating and Decimal Numbers

Floating and decimal numbers contain a decimal point and / or exponential notation. The following examples show floating and decimal numbers:

```
123.456     1.23456E2   123456.0E-3
```

The digits to the right of the decimal point in these examples are the decimal portions of the numbers.

The E that occurs in two of the examples is the symbol for exponential notation. The digit that follows E is the value of the exponent. For example, the number 3E5 (or 3E+5) means 3 multiplied by 10 to the fifth power, and the number 3E-5 means 3 multiplied by 10 to the minus fifth power.

Literal Numbers and the MONEY Data Type

When you use a literal number as a MONEY value, do not precede it with a money symbol or include commas.

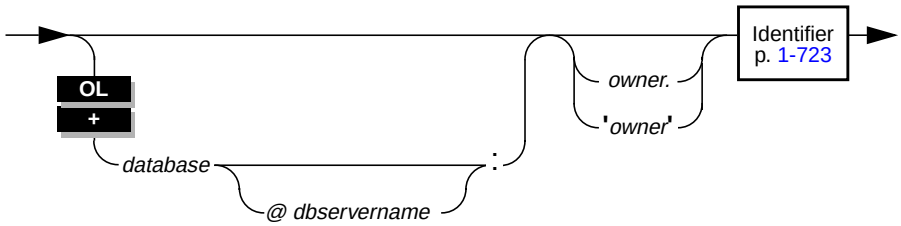
References

See the discussions of numeric data types, such as DECIMAL, FLOAT, INTEGER, and MONEY, in Chapter 3 of the *Informix Guide to SQL: Reference*.

Procedure Name

The Procedure Name segment specifies the name of a stored procedure. Use the Procedure Name segment whenever you see a reference to a procedure name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the procedure resides	The database must exist.	Database Name, p. 1-660
dbservername	The name of the OnLine database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-660
owner	The user name of the owner of the procedure	If you are using an ANSI-compliant database, you must specify an owner for a procedure you do not own. If you put quotation marks around the name you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The actual name of the procedure is an SQL identifier.

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of procedures. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

If you are creating the procedure, the *name* of the procedure must be unique within a database.

If you are creating the procedure, the combination *owner.name* must be unique within a database.

The owner name is case sensitive. In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the table owner is stored as uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-770. ♦

Procedures and SQL Functions with the Same Names

If you create a procedure with the same name as an SQL ledure winust be4.63 39.852 -

```
  LET d = length(5);
END

RETURN c, d;

END PROCEDURE;
```

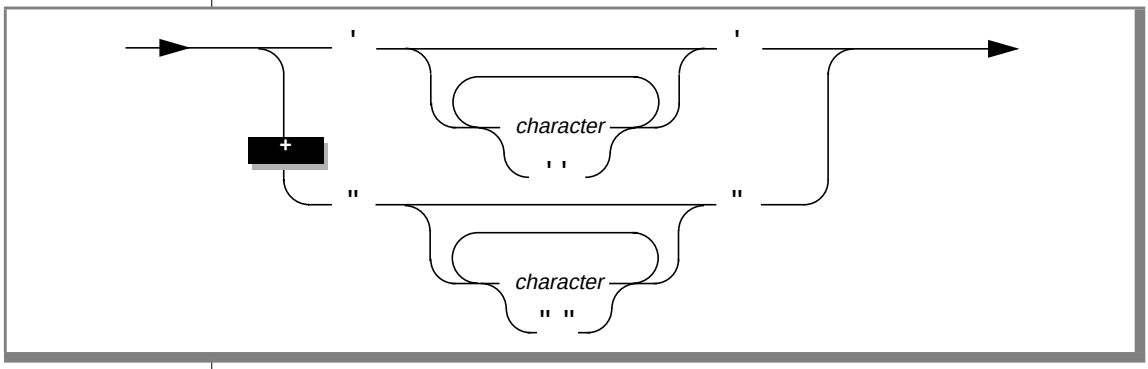
References

See the CREATE PROCEDUREee the

Quoted String

A quoted string is a string constant that is surrounded by quotation marks. Use the Quoted String segment whenever you see a reference to a quoted string in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
character	A character that forms part of the quoted string	The character or characters in the quoted string cannot be surrounded by double quotes if the **DELIMIDENT** environment variable is set. For additional restrictions see "Restrictions on Specifying Characters in Quoted Strings" on page 1-758.	Characters are literal values that you enter from the keyboard.

Restrictions on Specifying Characters in Quoted Strings

You must observe the following restrictions when you specify *character* in quoted strings:

- If you are using the ASCII code set, you can specify any printable ASCII character, including a single quote or double quote. For restrictions that apply to using quotes within quoted strings, see "Using Quotes in Strings" on page 1-759.

GLS

- If you are using a nondefault locale, you can specify non-ASCII characters, including multibyte characters, that the code set of your locale supports. See the discussion of quoted strings in Chapter 3 of the *Guide to GLS Functionality* for further information. ♦
- When you set the **DELIMIDENT** environment variable, you cannot use double quotes to delimit a quoted string. When **DELIMIDENT** is set, a string enclosed in double quotes is an identifier, not a quoted string. When **DELIMIDENT** is not set, a string enclosed in double quotes is a quoted string, not an identifier. See "Using Quotes in Strings" on page 1-759 for further information.
- You can enter DATETIME and INTERVAL data as quoted strings. See "DATETIME and INTERVAL Values as Strings" on page 1-759 for the restrictions that apply to entering DATETIME and INTERVAL data in quoted-string format.
- Quoted strings that are used with the LIKE or MATCHES keyword in a search condition can include wildcard characters that have a special meaning in the search condition. See "LIKE and MATCHES in a Condition" on page 1-759 for further information.
- When you insert a value that is a quoted string, you must observe a number of restrictions. See "Inserting Values as Quoted Strings" on page 1-760 for further information.

Usage

The string constant must be written on a single line; that is, you cannot use embedded new lines.

Using Quotes in Strings

The single quote has no special significance in string constants delimited by double quotes. Likewise, the double quote has no special significance in strings delimited by single quotes. For example, the following strings are valid:

```
"Nancy's puppy jumped the fence"
'Billy told his kitten, "no!"'
```

If your string is delimited by double quotes, you can include a double quote in the string by preceding the double quote with another double quote, as shown in the following string:

```
"Enter ""y"" to select this row"
```

When the **DELIMIDENT** environment variable is set, double quotes delimit identifiers, not strings. See "Delimited Identifiers" on page 1-726 for further information on delimited identifiers.

DATETIME and INTERVAL Values as Strings

You can enter DATETIME and INTERVAL data in the literal forms described in the "Literal DATETIME" and "Literal INTERVAL" segments beginning on pages 1-746 and 1-749, respectively, or you can enter them as quoted strings. Valid literals that are entered as character strings are converted automatically into DATETIME or INTERVAL values. The following INSERT statements use quoted strings to enter INTERVAL and DATETIME data:

```
INSERT INTO cust_calls(call_dtime) VALUES ('1993-5-4 10:12:11')

INSERT INTO manufact(lead_time) VALUES ('14')
```

The format of the value in the quoted string must exactly match the format specified by the qualifiers of the column. For the first case in the preceding example, **call_dtime** must be defined with the qualifiers YEAR TO MINUTE for the INSERT statement to be valid.

LIKE and MATCHES in a Condition

Quoted strings with the LIKE or MATCHES keyword in a condition can include wildcard characters. See the "Condition" segment beginning on page 1-643 for a complete description of how to use wildcard characters.

Inserting Values as Quoted Strings

If you are inserting a value that is a quoted string, you must adhere to the following conventions:

- Enclose CHAR, VARCHAR, NCHAR, NVARCHAR, DATE, DATETIME, and INTERVAL values in quotation marks.
- Set DATE values in the *mm/dd/yyy format*.
- You cannot insert strings longer than 256 bytes.
- Numbers with decimal values must contain a decimal point. You cannot use a comma as a decimal indicator.
- You cannot precede MONEY data with a dollar sign ($) or include commas.
- You can include NULL as a placeholder only if the column accepts null values.

References

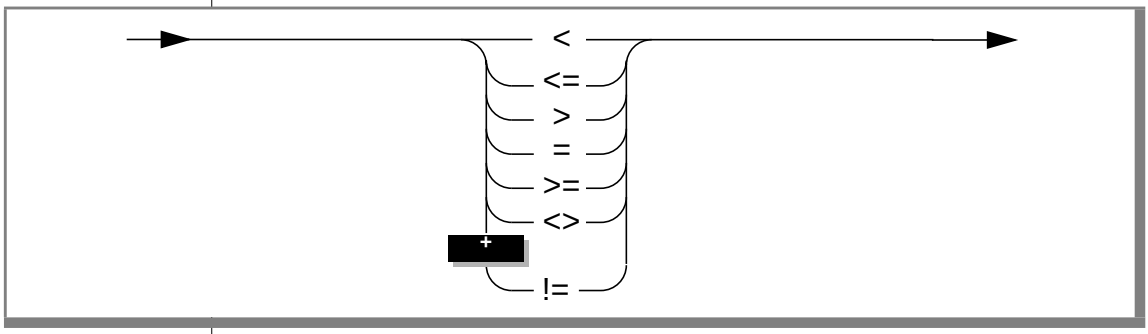
In the *Informix Guide to SQL: Reference*, see the discussion of the **DELIMIDENT** environment variable in Chapter 4.

In Chapter 3 of the *Guide to GLS Functionality*, see the discussion of quoted strings.

Relational Operator

A relational operator compares two expressions quantitatively. Use the Relational Operator segment whenever you see a reference to a relational operator in a syntax diagram.

Syntax



Each operator shown in the syntax diagram has a particular meaning.

Relational Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
=	Equal to
>=	Greater than or equal to
<>	Not equal to
!=	Not equal to

Usage

For DATE and DATETIME expressions, *greater than* means later in time.

For INTERVAL expressions, *greater than* means a longer span of time.

For CHAR and VARCHAR expressions, *greater than* means *after* in code-set order.

GLS

Locale-based collation order is used for NCHAR and NVARCHAR expressions. So for NCHAR and NVARCHAR expressions, *greater than* means *after* in the locale-based collation order. See Chapter 3 of the *Guide to GLS Functionality* for further information on locale-based collation order and the NCHAR and NVARCHAR data types. ♦

Collating Order for English Data

If you are using the default locale (U.S. English), the database server uses the code-set order of the default code set when it compares the character expressions that precede and follow the relational operator. On UNIX platforms, the default code set is the ISO8859-1 code set, which consists of the following sets of characters:

- The ASCII characters have code points in the range of 0 to 127.

  This range contains control characters, punctuation symbols, English-language characters, and numerals.
- The 8-bit characters have code points in the range 128 to 255.

  This range includes many non-English-language characters (such as é, â, ö, and ñ) and symbols (such as £, ©, and ¿).

The following table shows the ASCII code set. The Num column shows the ASCII code numbers, and the Char column shows the ASCII character corresponding to each ASCII code number. ASCII characters are sorted according to their ASCII code number. Thus lowercase letters follow uppercase letters, and both follow numerals. In this table, the caret symbol (^) stands for the CTRL key. For example, ^X means CTRL-X.

Num	Char	Num	Char	Num	Char
0	^@	43	+	86	V
1	^A	44	,	87	W
2	^B	45	-	88	X
3	^C	46	.	89	Y
4	^D	47	/	90	Z
5	^E	48	0	92	[
6	^F	49	1	93	\
7	^G	50	2	94	]
8	^H	51	3	94	^
9	^I	52	4	95	_
10	^J	53	5	96	`
11	^K	54	6	97	a
12	^L	55	7	98	b
13	^M	56	8	99	c
14	^N	57	9	100	d
15	^O	58	:	101	e
16	^P	59	;	102	f
17	^Q	60	<	103	g
18	^R	61	=	104	h
19	^S	62	>	105	i

(1 of 2)

Num	Char	Num	Char	Num	Char
20	^T	63	?	106	j
21	^U	64	@	107	k
22	^V	65	A	108	l
23	^W	66	B	109	m
24	^X	67	C	110	n
25	^Y	68	D	111	o
26	^Z	69	E	112	p
27	esc	70	F	113	q
28	^ \	71	G	114	r
29	^]	72	H	115	s
30	^^	73	I	116	t
31	^_	74	J	117	u
32		75	K	118	v
33	!	76	L	119	w
34	"	77	M	120	x
35	#	78	N	121	y
36	$	79	O	122	z
37	%	80	P	123	{
38	&	81	Q	124	|
39	'	82	R	125	}
40	(	83	S	126	~
41	)	84	T	127	del
42	*	85	U		

(2 of 2)

Support for ASCII Characters in Nondefault Code Sets

GLS

Most code sets in nondefault locales (called nondefault code sets) support the ASCII characters. If you are using a nondefault locale, the database server uses ASCII code-set order for any ASCII data in CHAR and VARCHAR expressions, as long as the nondefault code set supports these ASCII characters. ♦

References

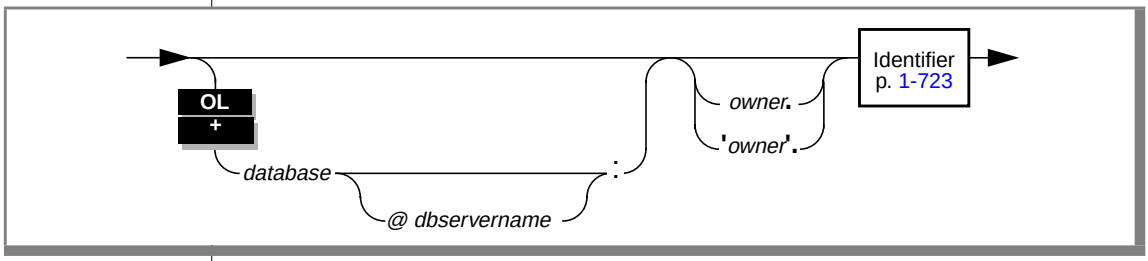
In the *Informix Guide to SQL: Tutorial*, see the discussion of relational operators in the SELECT statement in Chapter 2.

In Chapter 3 of the *Guide to GLS Functionality*, see the discussion of relational operator conditions in the SELECT statement.

Synonym Name

The Synonym Name segment specifies the name of a synonym. Use the Synonym Name segment whenever you see a reference to a synonym name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the synonym resides	The database must exist.	Database Name, p. 1-660
dbservername	The name of the OnLine database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-660
owner	The user name of the owner of the synonym	If you are using an ANSI-compliant database, you must specify the owner for a synonym that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The actual name of the synonym is an SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of synonyms. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

If you are creating the synonym, the *name* of the synonym must be unique within a database. The *name* cannot be the same as table names, temporary table names, or view names. It is possible to have a public and private synonym with the same name.

ANSI

If you are creating the synonym, the combination *owner.name* must be unique within a database.

The owner name is case sensitive. In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the table owner is stored in uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-770. ♦

References

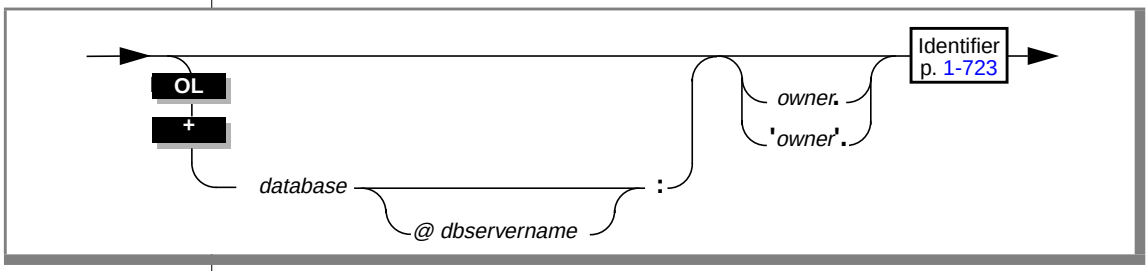
See the CREATE SYNONYM statement in this manual for information on creating synonyms.

In the *Informix Guide to SQL: Tutorial*, see the discussion of synonyms in Chapter 11.

Table Name

The Table Name segment specifies the name of a table. Use the Table Name segment whenever you see a reference to a table name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the table resides	The database must exist.	Database Name, p. 1-660
dbservername	The name of the OnLine database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-660
owner	The user name of the owner of the table	If you are using an ANSI-compliant database, you must specify the owner for a table that you do not own. If you put quotation marks around the name that you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters. In SELECT statements and other statements that access tables in an ANSI-compliant database, the table owner that you specify must exactly match the actual owner of the table. See "Case Sensitivity in ANSI-Compliant Databases" on page 1-770 for further information on this restriction.	The user name must conform to the conventions of your operating system.

Usage

The name of a table is an SQL identifier. The following example shows a table specification:

```
empinfo@personnel:emp_names
```

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of tables. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

If you are creating or renaming a table, the *name* of the table must be unique among all the tables, synonyms, temporary tables, and views that already exist in the database.

ANSI

If you are creating or renaming a table, you must make sure that the combination of *owner* and *name* is unique within a database.

In an ANSI-compliant database, the table name must include *owner.* unless you are the owner. For system catalog tables, the owner is **informix**. ♦

Case Sensitivity in ANSI-Compliant Databases

ANSI

The database server shifts the owner name to uppercase letters before the statement executes, unless the owner name is enclosed in quotes. Put quotes around the owner portion of a name if you want the owner to be read exactly as written. In the following example, the name **cathl** in the first statement is upshifted to **CATHL** before it is used; the name **nancy** in the second statement is not upshifted:

```
SELECT * FROM cathl.customer

SELECT * FROM 'nancy'.customer
```

No problem exists if you create a table with an implicit owner in uppercase letters and the owner's real login name is also in uppercase letters. For example, suppose that you are the user **BROWN,** and you create a view with the following statement:

```
CREATE VIEW newcust AS
    SELECT fname, lname FROM customer WHERE state = 'NJ'
```

You, **BROWN**, can run the following SELECT statements on the view:

```
SELECT * FROM brown.newcust

SELECT * FROM newcust

SELECT * FROM systables WHERE tabname = newcust
    AND owner = USER
```

In the first query in the preceding example, the database server automatically upshifts **brown** before the SELECT statement executes. In the second query, the database server returns the owner name **BROWN** already upshifted. In the third query, USER returns the login name as it is stored—in this case, in uppercase letters. If you are the user **nancy,** and you use the following statement, the resulting view has the name **NANCY.njcust**:

```
CREATE VIEW nancy.njcust AS
    SELECT fname, lname FROM customer WHERE state = 'NJ'
```

If you are **nancy,** and you use the following statement, the resulting view has the name **nancy.njcust**:

```
CREATE VIEW 'nancy'.njcust AS
    SELECT fname, lname FROM customer WHERE state = 'NJ'
```

The following SELECT statement fails because it tries to match the name **NANCY.njcust** to the actual owner and table name of **nancy.njcust**:

```
SELECT * FROM nancy.njcust
```

♦

References

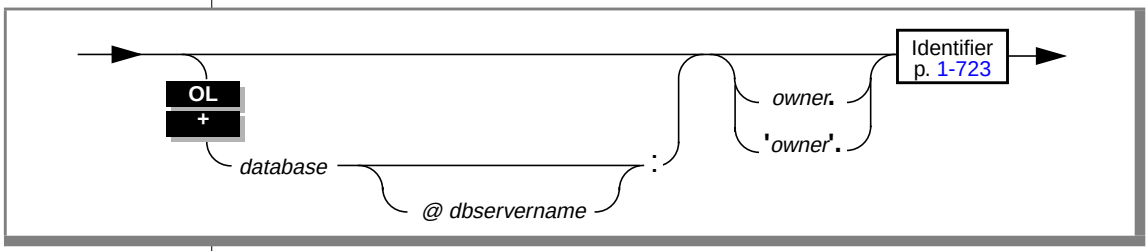
See the CREATE TABLE statement in this manual for information on creating tables.

In the *Informix Guide to SQL: Tutorial*, see the discussion of tables in Chapter 1.

View Name

The View Name segment specifies the name of a view. Use the View Name segment whenever you see a reference to a view name in a syntax diagram.

Syntax



Element	Purpose	Restrictions	Syntax
database	The name of the database where the view resides	The database must exist.	Database Name, p. 1-660
dbservername	The name of the OnLine database server that is home to *database*. The @ symbol is a literal character that introduces the database server name.	The database server that is specified in *dbservername* must match the name of a database server in the **sqlhosts** file.	Database Name, p. 1-660
owner	The user name of the owner of the view	If you are using an ANSI-compliant database, you must specify the owner for a view that you do not own. If you put quotation marks around the name you enter in *owner*, the name is stored exactly as typed. If you do not put quotation marks around the name that you enter in *owner*, the name is stored as uppercase letters.	The user name must conform to the conventions of your operating system.

Usage

The name of a view is an SQL identifier.

GLS

If you are using a nondefault locale, you can use characters from the code set of your locale in the names of views. For more information, see Chapter 3 of the *Guide to GLS Functionality*. ♦

The use of the prefix *owner.* is optional; however, if you use it, the database server does check *owner* for accuracy. If you are creating a view, the *name* of the view must be unique among all the tables, synonyms, temporary tables, and views that already exist in the database.

ANSI

If you are creating a view, the *owner.view-name* must be unique among all the tables, synonyms, and views that already exist in the database.

The owner name is case sensitive. In an ANSI-compliant database, if you do not use quotes around the owner name, the name of the table owner is stored as uppercase letters. For more information, see the discussion of case sensitivity in ANSI-compliant databases on page 1-770. ♦

References

See the CREATE VIEW statement in this manual for information on creating views.

In the *Informix Guide to SQL: Tutorial*, see the discussions of views in Chapter 10.

Chapter

2

SPL Statements

CALL 2-4
CONTINUE 2-7
DEFINE 2-8
EXIT 2-16
FOR. 2-18
FOREACH 2-23
IF 2-27
LET 2-31
ON EXCEPTION 2-34
RAISE EXCEPTION 2-39
RETURN 2-41
SYSTEM 2-43
TRACE 2-46
WHILE 2-49

You can use SQL statements and Stored Procedure Language (SPL) statements to write procedures, and you can store these procedures in the database. These stored procedures are effective tools for controlling SQL activity.

This chapter contains descriptions of the SPL statements. The description of each statement includes the following information:

- A brief introduction that explains the purpose of the statement
- A syntax diagram that shows how to enter the statement correctly
- A syntax table that explains each input parameter in the syntax diagram
- Rules of usage, including examples that illustrate these rules

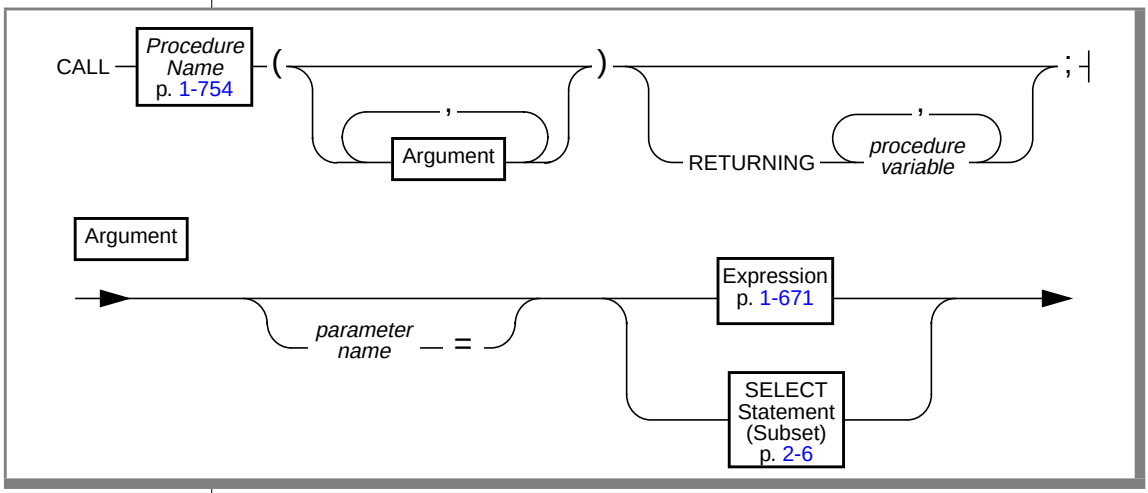
If a statement is composed of multiple clauses, the statement description provides the same set of information for each clause.

For task-oriented information about using stored procedures, see Chapter 12 of the *Informix Guide to SQL: Tutorial*.

CALL

Use the CALL statement to execute a procedure from within a stored procedure.

Syntax



Element	Purpose	Restrictions	Syntax
parameter name	The name of the parameter in the called procedure, as defined by its CREATE PROCEDURE statement	Name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use the *parameter name* = syntax for none or all the arguments that are specified in one CALL statement.	Identifier, p. 1-723
procedure variable	The name of a variable that receives the value being returned	The data type of *procedure variable* must match that of the value that is being returned.	Identifier, p. 1-723

Usage

The CALL statement invokes a procedure called *procedure name*. The CALL statement is identical in behavior to the EXECUTE PROCEDURE statement, but you can only use it from within a stored procedure.

Specifying Arguments

If CALL statement contains more arguments than the called procedure expects, you receive an error.

If a CALL statement specifies fewer arguments than the called procedure expects, the arguments are said to be missing. The database server initializes missing arguments to their corresponding default values. (See CREATE PROCEDURE on page 1-134.) This initialization occurs before the first executable statement in the body of the procedure.

If missing arguments do not have default values, the database server initializes the arguments to the value of UNDEFINED. An attempt to use any variable that has the value of UNDEFINED results in an error.

Either name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use the *parameter name* = syntax for all or none of the arguments that are specified in one CALL statement.

Each procedure call in the following example is valid for a procedure that expects character arguments `t`, `n`, and `d`, in that order:

```
CALL add_col (t='customer', n = 'newint', d ='integer');
CALL  add_col('customer','newint','integer');
```

Subset of SELECT Allowed in a Procedure Argument

You can use any SELECT statement as the argument for a procedure if it returns exactly one value of the proper data type and length. See the discussion of SELECT statements on page 1-459 for more information.

Receiving Input from the Called Procedure

The RETURNING clause specifies the *procedure variable* that receives the returned values from a procedure call. If you omit the RETURNING clause, the called procedure does not return any values.

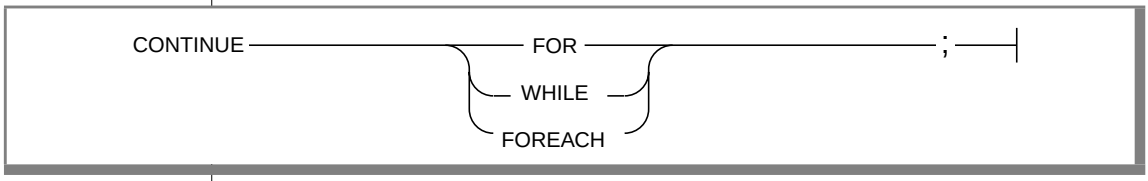
The following example shows two procedure calls, one that expects no returned values (**no_args**) and one that expects three returned values (**yes_args**). The creator of the procedure has defined three integer variables to receive the returned values from **yes_args**.

```
CREATE PROCEDURE not_much()
    DEFINE i, j, k INT;
    CALL no_args (10,20);
    CALL yes_args (5) RETURNING i, j, k;
END PROCEDURE
```

CONTINUE

Use the CONTINUE statement to start the next iteration of the innermost FOR, WHILE, or FOREACH loop.

Syntax



Usage

When you encounter a CONTINUE statement, the procedure skips the rest of the statements in the innermost loop of the indicated type. Execution continues at the top of the loop with the next iteration. In the following example, the procedure inserts values 3 through 15 into the table **testtable**. The procedure also returns values 3 through 9 and 13 through 15 in the process. The procedure does not return the value 11 because it encounters the CONTINUE FOR statement. The CONTINUE FOR statement causes the procedure to skip the RETURN i WITH RESUME statement.

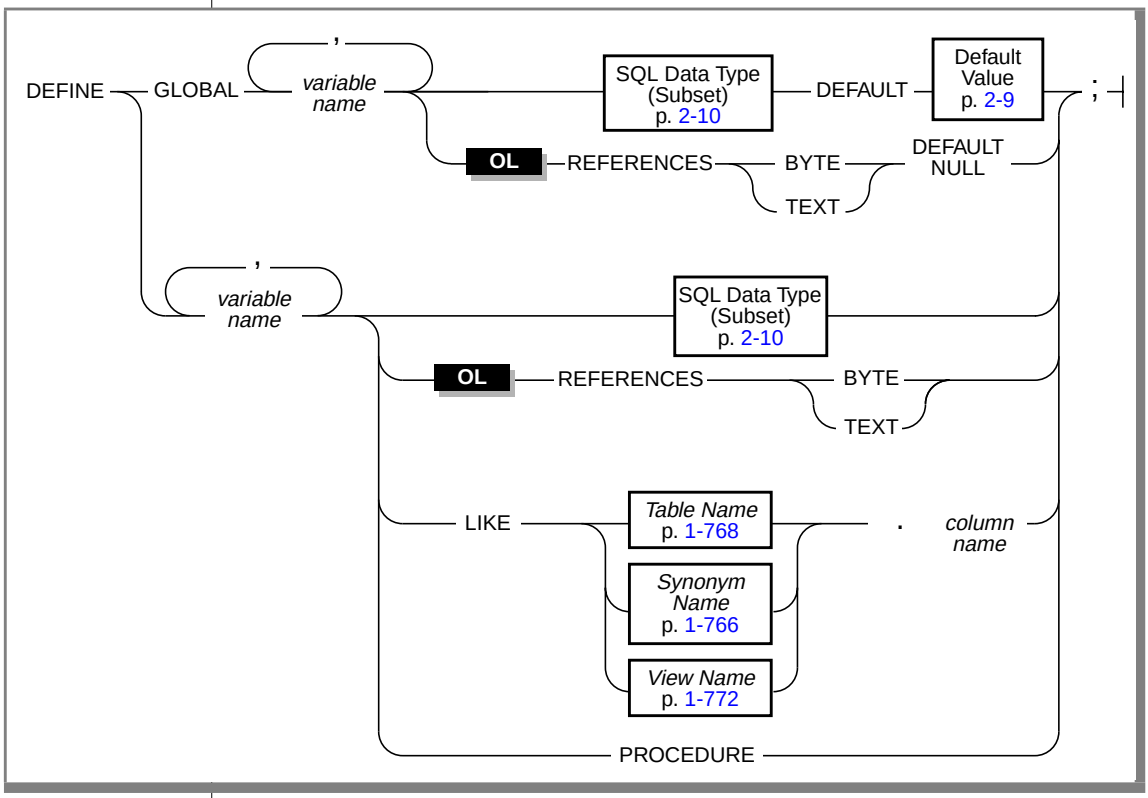
```
CREATE PROCEDURE loop_skip()
    RETURNING INT;
    DEFINE i INT;
    .
    .
    .
    FOR i IN (3 TO 15 STEP 2)
    INSERT INTO testtable values(i, null, null);
        IF i = 11
            CONTINUE FOR;
        END IF;
        RETURN i WITH RESUME;
    END FOR;

END PROCEDURE;
```

The CONTINUE statement generates errors if it cannot find the identified loop.

DEFINE

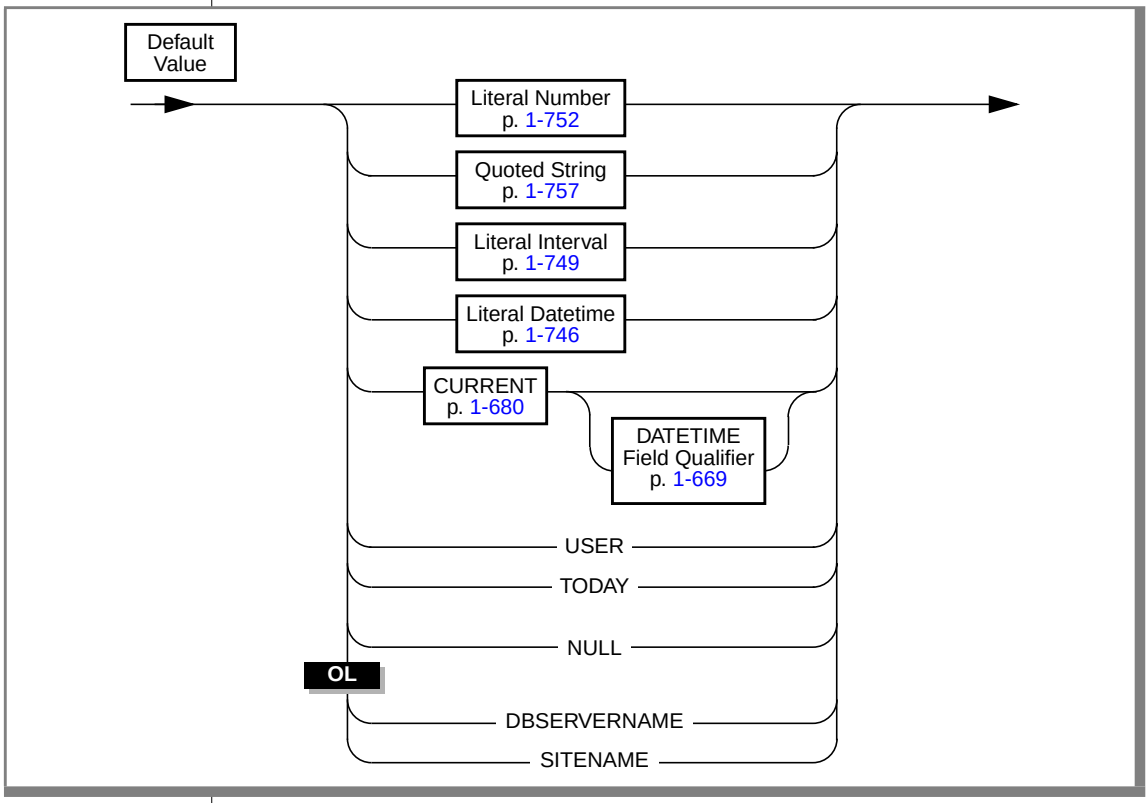
Use the DEFINE statement to declare variables that the procedure uses and to assign them data types.

Syntax



Element	Purpose	Restrictions	Syntax
column name	The name of a column in the table	The column must already exist in the table.	Identifier, p. 1-723
variable name	The name of the procedure variable that is being defined	The name must be unique to the statement block.	Identifier, p. 1-723

Default Value Clause



Usage

The DEFINE statement is not an executable statement. The DEFINE statement must appear after the procedure header and before any other statements. You can use a variable anywhere within the statement block where it is defined; that is, the scope of a defined variable is the statement block in which it was defined.

SQL Data Type Subset

The SQL data type subset includes all the SQL data types except SERIAL, TEXT, and BYTE.

Defining TEXT and BYTE Variables

The REFERENCES keyword lets you use TEXT and BYTE variables. TEXT and BYTE variables do not contain the actual data but are simply pointers to the data. The REFERENCES keyword is a reminder that the procedure variable is just a pointer. Use the procedure variables for TEXT and BYTE data types exactly as you would any other variable.

Redeclaration or Redefinition

If you define the same variable twice within the same statement block, you receive an error. You can redefine a variable within a nested block, in which case it temporarily hides the outer declaration. The following example produces an error:

```
CREATE PROCEDURE example1()
    DEFINE n INT; DEFINE j INT;
    DEFINE n CHAR (1); -- redefinition produces an error
    .
    .
    .
```

The database server allows the redeclaration in the following example. Within the nested statement block, `n` is a character variable. Outside the block, `n` is an integer variable.

```
CREATE PROCEDURE example2()
    DEFINE n INT; DEFINE j INT;
    .
    .
    .
    BEGIN
    DEFINE n CHAR (1); -- character n masks integer variable
                       -- locally
    .
    .
    .
END
```

Declaring GLOBAL Variables

The GLOBAL modifier indicates that the list of variables that follows the GLOBAL keyword are available to other procedures. The data types of these variables must match the data types of variables in the *global environment*. The global environment is the memory that is used by all the procedures that run within a given session (a DB-Access session or an SQL API session). The values of global variables are stored in memory.

Procedures that are running in the current session share global variables. Because the database server does not save global variables in the database, the global variables do not remain when the current session closes.

Databases do not share global variables. The database server and any application development tools do not share global variables.

The first declaration of a global variable establishes the variable in the global environment; subsequent global declarations simply bind the variable to the global environment and establish the value of the variable at that point. The following example shows two procedures, **proc1** and **proc2**; each has defined the global variable **gl_out**:

```
CREATE PROCEDURE proc1()
    .
    .
    .
    DEFINE GLOBAL gl_out INT DEFAULT 13;
    .
    .
    .
    LET gl_out = gl_out + 1;
    END PROCEDURE;

    CREATE PROCEDURE proc2()
    .
    .
    .
    DEFINE GLOBAL gl_out INT DEFAULT 23;
    DEFINE tmp INT;
    .
    .
    .
    LET tmp = gl_out
    .
    .
    .
END PROCEDURE;
```

If **proc1** is called first, **gl_out** is set to 13 and then incremented to 14. If **proc2** is then called, it sees that the value of **gl_out** is already defined, so the default value of 23 is not applied. Then, **proc2** assigns the existing value of 14 to **tmp**. If **proc2** had been called first, **gl_out** would have been set to 23, and 23 would have been assigned to **tmp**. Later calls to **proc1** would not apply the default of 13.

Providing Default Values

You can provide a literal value or a null value as the default for a global variable. You can also use a call to an SQL function to provide the default value. The following example uses the SITENAME function to provide a default value. It also defines a global BYTE variable.

```
CREATE PROCEDURE gl_def()
    DEFINE GLOBAL gl_site CHAR(18) DEFAULT SITENAME;
    DEFINE GLOBAL gl_byte REFERENCES BYTE DEFAULT NULL;
    .
    .
    .
END PROCEDURE
```

SITENAME or DBSERVERNAME

If you use the value returned by SITENAME or DBSERVERNAME as the default, the variable must be a CHAR, VARCHAR, NCHAR, or NVARCHAR value of at least 18 characters.

USER

If you use USER as the default, the variable must be a CHAR, VARCHAR, NCHAR, or NVARCHAR value of at least 8 characters.

CURRENT

If you use CURRENT as the default, the variable must be a DATETIME value. If the YEAR TO FRACTION keyword has qualified your variable, you can use CURRENT without qualifiers. If your variable uses another set of qualifiers, you must provide the same qualifiers when you use CURRENT as the default value. The following example defines a DATETIME variable with qualifiers and uses CURRENT with matching qualifiers:

```
DEFINE GLOBAL d_var DATETIME YEAR TO MONTH
        DEFAULT CURRENT YEAR TO MONTH;
```

TODAY

If you use TODAY as the default, the variable must be a DATE value.

TEXT and BYTE

The only default value possible for a TEXT or BYTE variable is null. The following example defines a TEXT global variable that is called **l_blob**:

```
CREATE PROCEDURE use_text()
    DEFINE i INT;
    DEFINE GLOBAL l_blob REFERENCES TEXT DEFAULT NULL;
END PROCEDURE
```

Declaring Local Variables

Nonglobal (local) variables do not allow defaults. The following example shows typical definitions of local variables:

```
CREATE PROCEDURE def_ex()
    DEFINE i INT;
    DEFINE word CHAR(15);
    DEFINE b_day DATE;
    DEFINE c_name LIKE customer.fname;
    DEFINE b_text REFERENCES TEXT ;
END PROCEDURE
```

Declaring Variables LIKE Columns

If you use the LIKE clause, the database server defines *variable name* as the same data type as the *column* in *table*. The data types of variables that are defined as database columns are resolved at runtime; therefore, *column* and *table* do not need to exist at compile time.

Declaring Variables as the PROCEDURE Type

The PROCEDURE type indicates that in the current scope, *variable name* is a user-defined procedure call and not an SQL function or a system function call. For example, the following statement defines **length** as a stored procedure, not as the SQL LENGTH function. This definition disables the SQL LENGTH function within the scope of the statement block. You would use such a definition if you had created a procedure with the name **length** before you defined and used it in another procedure, as shown in the following example:

```
DEFINE length PROCEDURE;
.
.
.
LET x = length (a,b,c)
```

If you create a procedure with the same name as an aggregate function (SUM, MAX, MIN, AVG, COUNT) or with the name **extend**, you must qualify the procedure name with the owner name.

Declaring Variables for BYTE and TEXT Data

The keyword REFERENCES indicates that *variable name* is not a BYTE or TEXT value but a pointer to the BYTE or TEXT value. Use the variable as though it holds the data.

The following example defines a local BYTE variable:

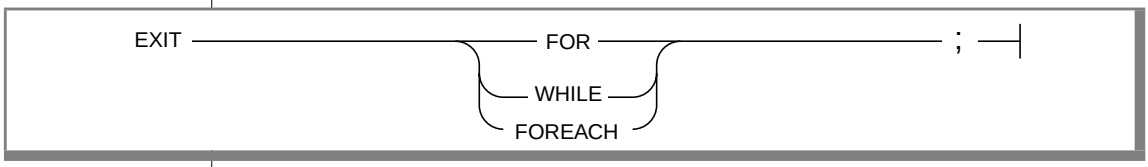
```
CREATE PROCEDURE use_blob()
    DEFINE i INT;
    DEFINE l_blob REFERENCES BYTE;
END PROCEDURE --use_blob
```

If you pass a variable of TEXT or BYTE data type to a procedure, the data is passed to the database server and stored in the root dbspace or dbspaces that the **DBSPACETEMP** environment variable specifies, if it is set. You do not need to know the location or name of the file that holds the data. BYTE or TEXT manipulation requires only the name of the BYTE or TEXT variable as it is defined in the procedure.

EXIT

Use the EXIT statement to stop the execution of a FOR, WHILE, or FOREACH loop.

Syntax



Usage

The EXIT statement causes the innermost loop of the indicated type (FOR, WHILE, or FOREACH) to terminate. Execution resumes at the first statement outside the loop.

If the EXIT statement cannot find the identified loop, it fails.

If the EXIT statement is used outside all loops, it generates errors.

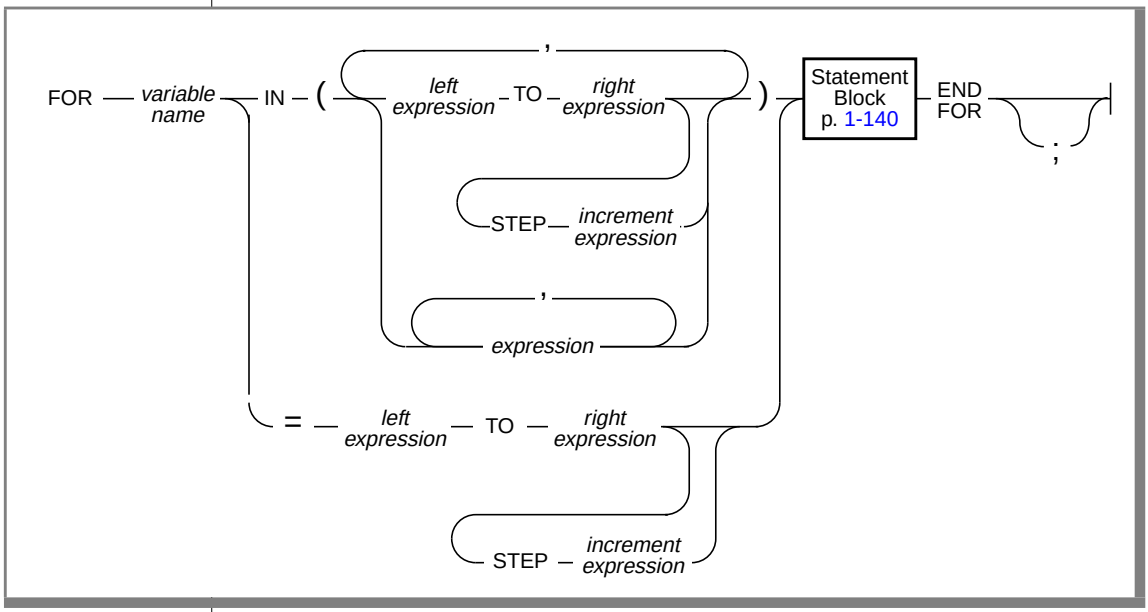
The following example uses an EXIT FOR statement. In the FOR loop, when j becomes 6, the IF condition i = 5 in the WHILE loop is true. The FOR loop stops executing, and the procedure continues at the next statement outside the FOR loop (in this case, the END PROCEDURE statement). In this example, the procedure ends when j equals 6.

```
CREATE PROCEDURE ex_cont_ex()
    DEFINE i,s,j, INT;

    FOR j = 1 TO 20
        IF j > 10 THEN
            CONTINUE FOR;
        END IF

        LET i,s = j,0;
        WHILE i > 0
            LET i = i -1;
            IF i = 5 THEN
                EXIT FOR;
            END IF
        END WHILE
    END FOR
END PROCEDURE
```

FOR

Use the FOR statement to initiate a controlled (definite) loop when you want to guarantee termination of the loop. The FOR statement uses expressions or range operators to establish a finite number of iterations for a loop.

Syntax



Element	Purpose	Restrictions	Syntax
expression	A numeric or character value against which *variable name* is compared to determine if the loop should be executed	The data type of *expression* must match the data type of *variable name*. You can use the output of a SELECT statement as an *expression*.	Expression, p. 1-671
increment expression	A positive or negative value by which *variable name* is incremented. Defaults to +1 or -1 depending on *left expression* and *right expression*.	The increment expression cannot evaluate to 0.	Expression, p. 1-671
left expression	The starting expression of a range	The value of *left expression* must match the data type of *variable name*. It must be either INT or SMALLINT.	Expression, p. 1-671
right expression	The ending expression in the range. The size of *right expression* relative to *left expression* determines if the range is stepped through positively or negatively.	The value of *right expression* must match the data type of *variable name*. It must be either INT or SMALLINT.	Expression, p. 1-671
variable name	The value of this variable determines how many times the loop executes.	You must have already defined this variable, and the variable must be valid within this statement block. If you are using *variable name* with a range of values and the TO keyword, you must define *variable name* explicitly as either INT or SMALLINT.	Identifier, p. 1-723

Usage

The database server computes all expressions before the FOR statement executes. If one or more of the expressions are variables, and their values change during the loop, the change has no effect on the iterations of the loop.

The FOR loop terminates when *variable name* takes on the values of each element in the expression list or range in succession or when it encounters an EXIT FOR statement.

The database server generates an error if an assignment within the body of the FOR statement attempts to modify the value of *variable name*.

Using the TO Keyword to Define a Range

The TO keyword implies a range operator. The range is defined by *left expression* and *right expression*, and the STEP *increment expression* option implicitly sets the number of increments. If you use the TO keyword, *variable name* must be an INT or SMALLINT data type. The following example shows two equivalent FOR statements. Each uses the TO keyword to define a range. The first statement uses the IN keyword, and the second statement uses an equal sign (=). Each statement causes the loop to execute five times.

```
FOR index_var IN (12 TO 21 STEP 2)
    -- statement block
END FOR

FOR index_var = 12 TO 21 STEP 2
    -- statement block
END FOR
```

If you omit the STEP option, the database server gives *increment expression* the value of -1 if *right expression* is less than *left expression*, or +1 if *right expression* is more than *left expression*. If *increment expression* is specified, it must be negative if *right expression* is less than *left expression*, or positive if *right expression* is more than *left expression*. The two statements in the following example are equivalent. In the first statement, the STEP increment is explicit. In the second statement, the STEP increment is implicitly 1.

```
FOR index IN (12 TO 21 STEP 1)
    -- statement block
END FOR

FOR index = 12 TO 21
    -- statement block
END FOR
```

The database server initializes the value of *variable name* to the value of *left expression*. In subsequent iterations, the server adds *increment expression* to the value of *variable name* and checks *increment expression* to determine whether the value of *variable name* is still between *left expression* and *right expression*. If so, the next iteration occurs. Otherwise, an exit from the loop takes place. Or, if you specify another range, the variable takes on the value of the first element in the next range.

Specifying Two or More Ranges in a Single FOR Statement

The following example shows a statement that traverses a loop forward and backward and uses different increment values for each direction:

```
FOR index_var IN (15 to 21 STEP 2, 21 to 15 STEP -3)
    -- statement body
END FOR
```

Using an Expression List as the Range

The database server initializes the value of *variable name* to the value of the first expression specified. In subsequent iterations, *variable name* takes on the value of the next expression. When the server has evaluated the last expression in the list and used it, the loop stops.

The expressions in the IN list do not have to be numeric values, as long as you do not use range operators in the IN list. The following example uses a character expression list:

```
FOR c IN ('hello', (SELECT name FROM t), 'world', v1, v2)
    INSERT INTO t VALUES (c);
    END FOR
```

The following FOR statement shows the use of a numeric expression list:

```
FOR index IN (15,16,17,18,19,20,21)
    -- statement block
END FOR
```

Mixing Range and Expression Lists in the Same FOR Statement

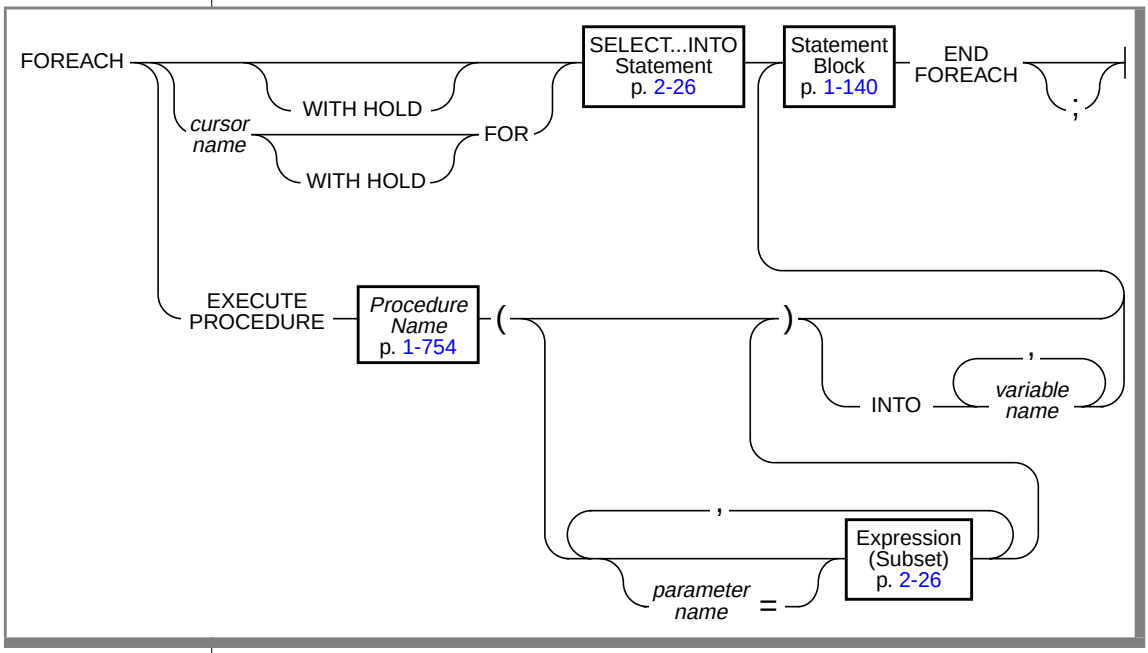
If *variable name* is an INT or SMALLINT value, you can mix ranges and expression lists in the same FOR statement. The following example shows a mixture that uses an integer variable. Values in the expression list include the value that is returned from a SELECT statement, a sum of an integer variable and a constant, the values that are returned from a procedure named **p_get_int**, and integer constants.

```
CREATE PROCEDURE for_ex ()
    DEFINE i, j INT;
    LET j = 10;
    FOR i IN (1 TO 20, (SELECT c1 FROM tab WHERE id = 1),
    j+20 to j-20, p_get_int(99),98,90 to 80 step -2)
        INSERT INTO tab VALUES (i);
    END FOR
END PROCEDURE
```

FOREACH

Use a FOREACH loop to select and manipulate more than one row.

Syntax



Element	Purpose	Restrictions	Syntax
cursor name	An identifier that you supply as a name for the SELECT...INTO statement	Each cursor name within a procedure must be unique.	Identifier, p. 1-723
parameter name	The name of a parameter in the procedure that is being executed as defined in its CREATE PROCEDURE statement	Name or position, but not both, binds procedure arguments to procedure parameters. You can use *parameter name* = syntax for all or none of the arguments that are specified in one FOREACH EXECUTE PROCEDURE statement.	Identifier, p. 1-723
variable name	The name of a procedure variable in the calling or executing procedure	The data type of *variable name* must be appropriate for the value that is being returned.	Identifier, p. 1-723

Usage

A FOREACH loop is the procedural equivalent of using a cursor. When a FOREACH statement executes, the database server takes the following actions:

1. It declares and implicitly opens a cursor.
2. It obtains the first row from the query that is contained within the FOREACH loop, or it obtains the first set of values from the called procedure.
3. It assigns each variable in the variable list the value of the corresponding value from the active set that the SELECT statement or the called procedure creates.
4. It executes the statement block.
5. It fetches the next row from the SELECT statement or called procedure on each iteration, and it repeats step 3.
6. It terminates the loop when it finds no more rows that satisfy the SELECT statement or called procedure. It closes the implicit cursor when the loop terminates.

Because the statement block can contain additional FOREACH statements, cursors can be nested. No limit exists to the number of cursors that can be nested.

A procedure that returns more than one row or set of values is called a *cursory procedure*.

The following procedure illustrates the three types of FOREACH statements: with a SELECT...INTO clause, with an explicitly named cursor, and with a procedure call:

```
CREATE PROCEDURE foreach_ex()
    DEFINE i, j INT;

    FOREACH SELECT c1 INTO i FROM tab order by 1
        INSERT INTO tab2 VALUES (i);
    END FOREACH

    FOREACH cur1 FOR SELECT c2, c3 INTO i, j FROM tab
        IF j > 100 THEN
            DELETE FROM tab WHERE CURRENT OF cur1;
            CONTINUE FOREACH;
        END IF
        UPDATE tab SET c2 = c2 + 10 WHERE CURRENT OF cur1;
    END FOREACH

    FOREACH EXECUTE PROCEDURE bar(10,20) INTO i
        INSERT INTO tab2 VALUES (i);
    END FOREACH
END PROCEDURE -- foreach_ex
```

A select cursor is closed when any of the following situations occur:

- The cursor returns no further rows.
- The cursor is a select cursor without a HOLD specification, and a transaction completes using COMMIT or ROLLBACK statements.
- An EXIT statement executes, which transfers control out of the FOREACH statement.
- An exception occurs that is not trapped inside the body of the FOREACH statement. (See the ON EXCEPTION statement on page 2-34.)
- A cursor in the calling procedure that is executing this cursory procedure (within a FOREACH loop) closes for any reason.

Using a SELECT...INTO Statement

The SELECT statement in the FOREACH statement must include the INTO clause. It can also include UNION and ORDER BY clauses, but it cannot use the INTO TEMP clause. The syntax of a SELECT statement is shown on page 1-459.

The type and count of each variable in the variable list must match each value that the SELECT...INTO statement returns.

Hold Cursors

The WITH HOLD keyword specifies that the cursor should remain open when a transaction closes (is committed or rolled back).

Updating or Deleting Rows Identified by Cursor Name

Use the WHERE CURRENT OF *cursor name* clause to update or delete the current row of *cursor name*.

Calling a Procedure in the FOREACH Loop

The called procedure can return zero or more rows.

The type and count of each variable in the variable list must match each value that the called procedure returns.

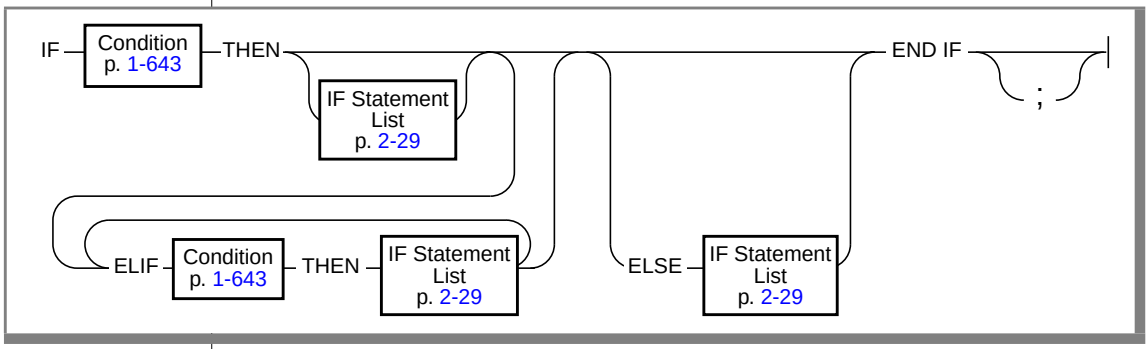
Subset of Expressions Allowed in the Procedure Parameters

You can use any expression as a procedure parameter except an aggregate expression. If you use a subquery or procedure call, the subquery or procedure must return a single value of the appropriate data type and size. For the full syntax of an expression, see page 1-671.

IF

Use an IF statement to create a branch within a procedure.

Syntax



Usage

The condition that the IF clause states is evaluated. If the result is true, the statements that follow the THEN keyword execute. If the result is false, and an ELIF clause exists, the statements that follow the ELIF clause execute. If no ELIF clause exists, or if the condition in the ELIF clause is not true, the statements that follow the ELIF keyword execute.

In the following example, the procedure uses an IF statement with both an ELIF clause and an ELSE clause. The IF statement compares two strings and displays a 1 to indicate that the first string comes before the second string alphabetically, or a -1 if the first string comes after the second string alphabetically. If the strings are the same, a 0 is returned.

```
CREATE PROCEDURE str_compare (str1 CHAR(20), str2 CHAR(20))
    RETURNING INT;
    DEFINE result INT;

    IF str1 > str2 then
        result =1;
    ELIF str2 > str1 THEN
```

```
            result = -1;
        ELSE
            result = 0;
        END IF
        RETURN result;
    END PROCEDURE -- str_compare
```

The ELIF Clause

Use the ELIF clause to specify one or more additional conditions to evaluate.

If you specify an ELIF clause, and the IF condition is false, the ELIF condition is evaluated. If the ELIF condition is true, the statements that follow the ELIF clause execute.

The ELSE Clause

The ELSE clause executes if no true previous condition exists in the IF clause or any of the ELIF clauses.

Conditions in an IF Statement

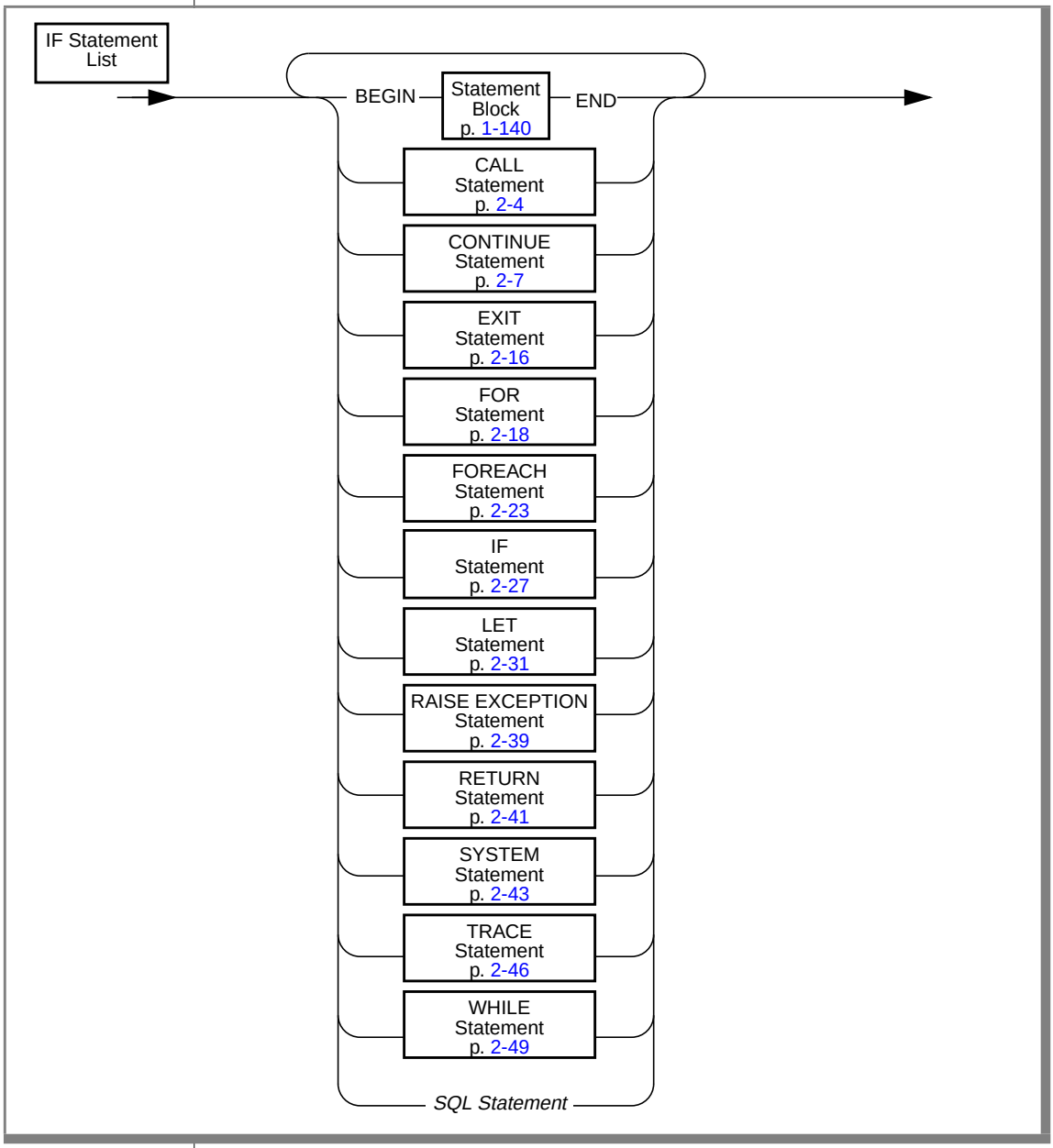
Conditions in an IF statement are evaluated in the same way as conditions in a WHILE statement.

If any expression that the condition contains evaluates to null, the condition automatically becomes untrue. Consider the following points:

1. If the expression *x* evaluates to null, then *x* is not true by definition. Furthermore, `not(x)` is also *not* true.
2. IS NULL is the sole operator that can yield true for *x*. That is, *x* IS NULL is true, and *x* IS NOT NULL is not true**.**

An expression within the condition that has an UNKNOWN value (due to the use of an uninitialized variable) causes an immediate error. The statement terminates and raises an exception.

IF Statement List



Subset of SQL Statements Allowed in an IF Statement

You can use any SQL statement in the statement block except the ones in the following list.

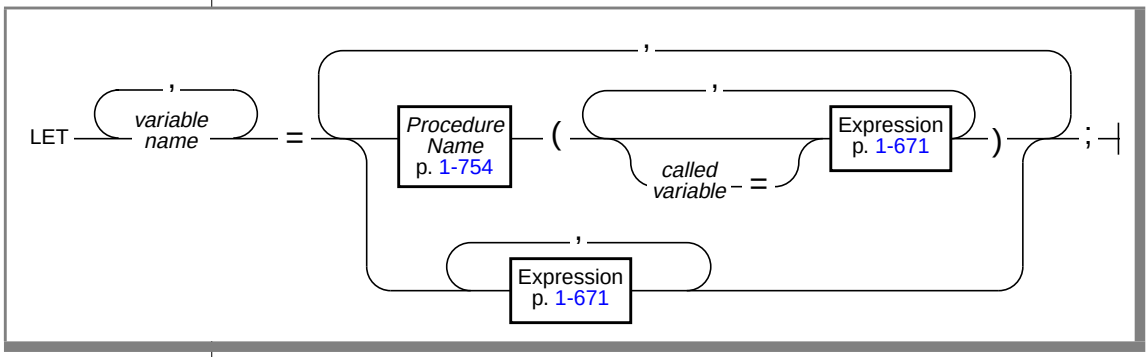
ALLOCATE DESCRIPTOR	GET DESCRIPTOR
CHECK TABLE	GET DIAGNOSTICS
CLOSE DATABASE	INFO
CONNECT	LOAD
CREATE DATABASE	OPEN
CREATE PROCEDURE	OUTPUT
DATABASE	PREPARE
DEALLOCATE DESCRIPTOR	PUT
DECLARE	REPAIR TABLE
DESCRIBE	ROLLFORWARD DATABASE
DISCONNECT	SET CONNECTION
EXECUTE	SET DESCRIPTOR
EXECUTE IMMEDIATE	START DATABASE
FETCH	UNLOAD
FLUSH	WHENEVER
FREE	

You can use a SELECT statement only if you use the INTO TEMP clause to put the results of the SELECT statement into a temporary table.

LET

Use the LET statement to assign values to variables. You also can use the LET statement to call a procedure within a procedure and assign the returned values to variables.

Syntax



Element	Purpose	Restrictions	Syntax
called variable	A procedure variable of the called procedure	Name or position, but not both, binds procedure arguments to procedure parameters. That is, you can use *called variable*= syntax for all or none of the arguments that are specified in a LET statement.	Identifier, p. 1-723
variable name	A procedure variable	The procedure variable must be defined in the procedure and valid in the statement block.	Identifier, p. 1-723

Usage

If you assign a value to a single variable, it is called a *simple assignment*; if you assign values to two or more variables, it is called a *compound assignment*.

At runtime, the value of the SPL expression is computed first. The resulting value is converted to *variable name* data type, if possible, and the assignment occurs. If conversion is not possible, an error occurs, and the value of *variable name* is undefined.

A compound assignment assigns multiple expressions to multiple variables. The count and data type of expressions in the expression list must match the count and data type of the corresponding variables in the variable list.

The following example shows several LET statements that assign values to procedure variables:

```
LET a   = c + d ;
LET a,b = c,d ;
LET expire_dt = end_dt + 7 UNITS DAY;
LET name = 'Brunhilda';
LET sname = DBSERVERNAME;
LET this_day = TODAY;
```

You cannot use multiple values to operate on other values. For example, the following statement is illegal:

```
LET a,b = (c,d) + (10,15); -- ILLEGAL EXPRESSION
```

Using a SELECT Statement in a LET Statement

Using a SELECT statement in a LET statement is equivalent to using a SELECT...INTO *procedure-variable* statement in a procedure. The examples in this section use a SELECT statement in a LET statement. You can use a SELECT statement to assign values to one or more variables on the left side of the = operator, as the following example shows:

```
LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
LET a,b,c = (SELECT c1,c2 FROM t WHERE id = 1), 15;
```

You cannot use a SELECT statement to make multiple values operate on other values. The code in the following example is illegal:

```
LET a,b = (SELECT c1,c2 FROM t) + (10,15); -- ILLEGAL CODE
```

Because a LET statement is equivalent to a SELECT...INTO statement, the two statements in the following example have the same results: `a=c` and `b=d`:

```
CREATE PROCEDURE proof()
    DEFINE a, b, c, d INT;
    LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
    SELECT c1, c2 INTO c, d FROM t WHERE id = 1
END PROCEDURE
```

If the SELECT statement returns more than one row, you must enclose the SELECT statement in a FOREACH loop.

Calling a Procedure in a LET Statement

You can call a procedure in a LET statement and assign the returned values to variables. If the LET statement includes a procedure call, it invokes the named procedure. You must specify all the necessary arguments to the procedure in the LET statement unless the procedure has default values for its arguments.

If you use the *called variable* = syntax for one of the parameters in the called procedure, you must use it for all the parameters.

The variable name receives the returned value from a procedure call. A procedure can return more than one value into a list of variable names. You must enclose a procedure that returns more than one row in a FOREACH loop.

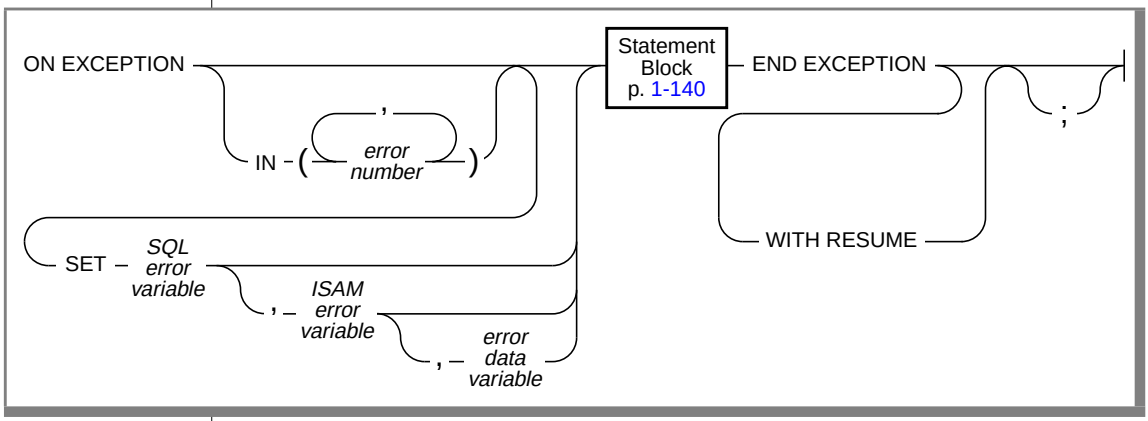
The following example shows several LET statements. The first two are valid LET statements that contain procedure calls. The third LET statement is not legal because it tries to add the output of two procedures and then assign the sum to two variables, *a* and *b*.You can easily split this LET statement into two legal LET statements.

```
LET a, b, c = proc1(name = 'grok', age = 17);
LET a, b, c = 7, proc ('orange', 'green');
LET a, b = proc1() + proc2(); -- ILLEGAL CODE
```

ON EXCEPTION

Use the ON EXCEPTION statement to specify the actions that are taken for a particular error or a set of errors.

Syntax



Element	Purpose	Restrictions	Syntax
error data variable	A procedure variable that contains a string returned by an SQL error or a user-defined exception	Must be a character data type to receive the error information. Must be valid in the current statement block.	Identifier, p. 1-723
error number	An SQL error number, or an error number created by a RAISE EXCEPTION statement, that is to be trapped	Must be of integer data type. Must be valid in the current statement block.	Literal number, p. 1-752
ISAM error variable	A variable that receives the ISAM error number of the exception raised	Must be of integer data type. Must be valid in the current statement block.	Identifier, p. 1-723
SQL error variable	A variable that receives the SQL error number of the exception raised	Must be a character data type. Must be valid in the current statement block.	Identifier, p. 1-723

Usage

The ON EXCEPTION statement, together with the RAISE EXCEPTION statement, provides an error-trapping and error-recovery mechanism for SPL. The ON EXCEPTION statement defines a list of errors that are to be trapped as the stored procedure executes and specifies the action (within the statement block) to take when the trap is triggered. If the IN clause is omitted, all errors are trapped.

You can use more than one ON EXCEPTION statement within a given statement block.

The scope of an ON EXCEPTION statement is the statement block that follows the ON EXCEPTION statement, all the statement blocks that are nested within that following statement block, and all the statement blocks that follow the ON EXCEPTION statement.

The exceptions that are trapped can be either system- or user-defined.

When an exception is trapped, the error status is cleared.

If you specify a variable to receive an ISAM error, but no accompanying ISAM error exists, a zero returns to the variable. If you specify a variable to receive the returned error text, but none exists, an empty string goes into the variable.

Placement of the ON EXCEPTION Statement

ON EXCEPTION is a declarative statement, not an executable statement. For this reason, you must use the ON EXCEPTION statement before any executable statement and after any DEFINE statement in a procedure.

The following example shows the correct placement of an ON EXCEPTION statement. Use an ON EXCEPTION statement after the DEFINE statement and before the body of the procedure. The following procedure inserts a set of values into a table. If the table does not exist, it is created, and the values are inserted. The procedure also returns the total number of rows in the table after the insert occurs.

```
CREATE PROCEDURE add_salesperson(last CHAR(15),
                 first CHAR(15))
    RETURNING INT;
    DEFINE x INT;
    ON EXCEPTION IN (-206) -- If no table was found, create one
        CREATE TABLE emp_list
             (lname CHAR(15),fname CHAR(15), tele CHAR(12));
        INSERT INTO emp_list VALUES -- and insert values
             (last, first, '800-555-1234');
    END EXCEPTION WITH RESUME
    INSERT INTO emp_list VALUES (last, first, '800-555-1234')
    LET x = SELECT count(*) FROM emp_list;
    RETURN x;
END PROCEDURE
```

When an error occurs, the database server searches for the last declaration of the ON EXCEPTION statement, which traps the particular error code. The ON EXCEPTION statement can have the error number in the IN clause or have no IN clause. If the database server finds no pertinent ON EXCEPTION statement, the error code passes back to the caller (the procedure, application, or interactive user), and execution aborts.

The following example uses two ON EXCEPTION statements with the same error number so that error code 691 can be trapped in two levels of nesting:

```
CREATE PROCEDURE delete_cust (cnum INT)
    ON EXCEPTION  IN (-691)    -- children exist
        BEGIN -- Begin-end is necessary so that other DELETEs
             -- don't get caught in here.
            ON EXCEPTION IN (-691)
                DELETE FROM another_child WHERE num = cnum;
                DELETE FROM orders WHERE customer_num = cnum;
            END EXCEPTION -- for 691

        DELETE FROM orders WHERE customer_num = cnum;
        END

      DELETE FROM cust_calls WHERE customer_num = cnum;
      DELETE FROM customer WHERE customer_num = cnum;
    END EXCEPTION
       DELETE FROM customer WHERE customer_num = cnum;
END PROCEDURE
```

Using the IN Clause to Trap Specific Exceptions

A trap is triggered if either the SQL error code or the ISAM error code matches an exception code in the list of error numbers. The search through the list begins from the left and stops with the first match.

You can use a combination of an ON EXCEPTION statement without an IN clause and one or more ON EXCEPTION statements with an IN clause to set up default trapping. A summary of the sequence of statements in the following example would be: "Test for an error. If error -210, -211, or -212 occurs, take action A. If error -300 occurs, take action B. If any other error occurs, take action C." When an error occurs, the database server searches for the last declaration of the ON EXCEPTION statement that traps the particular error code.

```
CREATE PROCEDURE ex_test ()
    DEFINE error_num INT;
    .
    .
    .
    ON EXCEPTION
    SET error_num
    -- action C
    END EXCEPTION

    ON EXCEPTION IN (-300)
    -- action B
    END EXCEPTION
    ON EXCEPTION IN (-210, -211, -212)
    SET error_num
    -- action A
    END EXCEPTION
    .
    .
    .
```

Receiving Error Information in the SET Clause

If you use the SET clause, when an exception occurs, the SQL error code and (optionally) the ISAM error code are inserted into the variables that are specified in the SET clause. If you provided an *error data variable*, any error text that the database server returns is put into the *error data variable*. Error text includes information such as the offending table or column name.

Forcing Continuation of the Procedure

The example on page 2-36 uses the WITH RESUME keyword to indicate that after the statement block in the ON EXCEPTION statement executes, execution is to continue at the `LET x = SELECT COUNT(*) FROM emp_list` statement, which is the line following the line that raised the error. For this procedure, the result is that the count of salespeople names occurs even if the error occurred.

Continuing Execution After an Exception Occurs

If you do not include the WITH RESUME keyword in your ON EXCEPTION statement, the next statement that executes after an exception occurs depends on the placement of the ON EXCEPTION statement, as the following scenarios describe:

- If the ON EXCEPTION statement is inside a statement block with a BEGIN and an END keyword, execution resumes with the first statement (if any) after that BEGIN...END block. That is, it resumes after the scope of the ON EXCEPTION statement.
- If the ON EXCEPTION statement is inside a loop (FOR, WHILE, FOREACH), the rest of the loop is skipped, and execution resumes with the next iteration of the loop.
- If no statement or block, but only the procedure, contains the ON EXCEPTION statement, the procedure executes a RETURN statement with no arguments to terminate. That is, the procedure returns a successful status and no values.

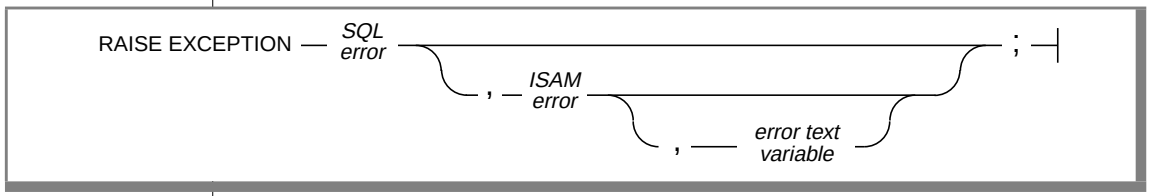
Errors Within the ON EXCEPTION Statement Block

To prevent an infinite loop, if an error occurs during execution of the statement block of an error trap, the search for another trap does not include the current trap.

RAISE EXCEPTION

Use the RAISE EXCEPTION statement to simulate the generation of an error.

Syntax



Element	Purpose	Restrictions	Syntax
error text variable	A procedure variable that contains the error text	The procedure variable must be a character data type and must be valid in the statement block.	Identifier, p. 1-723
ISAM error	A variable or expression that represents an ISAM error number. The default value is 0.	The variable or expression must evaluate to a SMALLINT value. You can place a minus sign before the error number.	Expression, p. 1-671
SQL error	A variable or expression that represents an SQL error number	The variable or expression must evaluate to a SMALLINT value. You can place a minus sign before the error number.	Expression, p. 1-671

Usage

Use the RAISE EXCEPTION statement to simulate an error. An ON EXCEPTION statement can trap the generated error.

If you omit the *ISAM error* parameter, the database server sets the ISAM error code to zero when the exception is raised. If you want to use the *error text variable* parameter but not specify a value for *ISAM error*, you can specify 0 as the value of *ISAM error*.

For example, the following statement raises the error number -9999 and returns the stated text:

```
RAISE EXCEPTION -9999, 0, 'You broke the rules';
```

The statement can raise either system-generated exceptions or user-generated exceptions.

In the following example, a negative value for a raises exception 9999. The code should contain an ON EXCEPTION statement that traps for an exception of 9999.

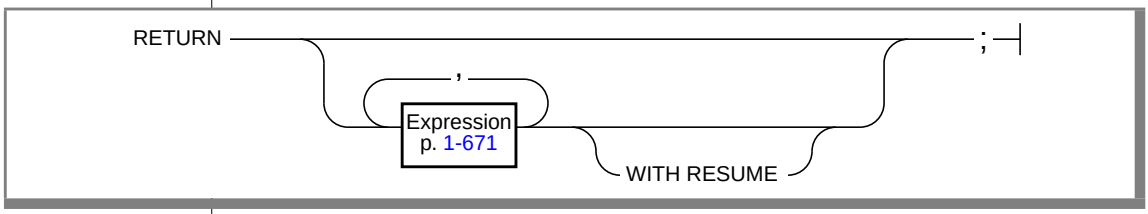
```
FOREACH SELECT c1 INTO a FROM t
IF a < 0 THEN
RAISE EXCEPTION 9999-- emergency exit
END IF
END FOREACH
```

See the ON EXCEPTION statement for more information about the scope and compatibility of exceptions.

RETURN

Use the RETURN statement to designate the values that the procedure returns to the calling module.

Syntax



Usage

The RETURN statement returns zero or more values to the calling process.

All the RETURN statements in the procedure must be consistent with the RETURNING clause of the CREATE PROCEDURE statement, which the procedure defines. The number and data type of values in the RETURN statement, if any, must match in number and data type the data types that are listed in the RETURNING clause of the CREATE PROCEDURE statement. You can choose to return no values even if you specify one or more values in the RETURNING clause. If you use a RETURN statement without any expressions, but the calling procedure or program expects one or more return values, it is equivalent to returning the expected number of null values to the calling program.

In the following example, the procedure includes two acceptable RETURN statements. A program that calls this procedure should check if no values are returned and act accordingly.

```
CREATE PROCEDURE two_returns (stockno INT)
    RETURNING CHAR (15);
    DEFINE des CHAR(15);
    ON EXCEPTION (-272)  -- if user doesn't have select privs...
        RETURN;    -- return no values.
    END EXCEPTION;
    SELECT DISTINCT descript INTO des FROM stock
            WHERE stocknum = stockno;
    RETURN des;
END PROCEDURE
```

A RETURN statement without any expressions exits only if the procedure is declared not to return values; otherwise it returns nulls.

The WITH RESUME Keyword

If you use the WITH RESUME keyword after the RETURN statement executes, the next invocation of this procedure (upon the next FETCH or FOREACH statement) starts from the statement that follows the RETURN statement. If a procedure executes a RETURN WITH RESUME statement, a FOREACH loop in the calling procedure or program must call the procedure.

ESQL

If a procedure executes a RETURN WITH RESUME statement, a FETCH statement in an application that is written in an SQL API can call it. ♦

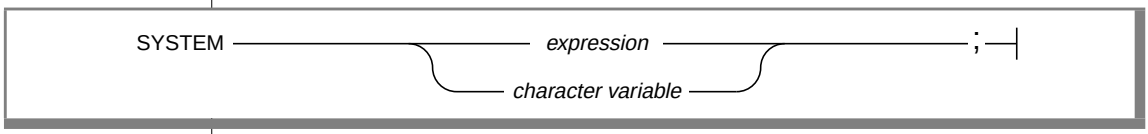
The following example shows a cursory procedure that another procedure can call. After the RETURN i WITH RESUME statement returns each value to the calling procedure, the next line of **sequence** executes the next time **sequence** is called. If **backwards** equals 0, no value is returned to the calling procedure, and execution of **sequence** stops.

```
CREATE PROCEDURE sequence (limit INT, backwards INT)
    RETURNING INT;
    DEFINE i INT;

    FOR i IN (1 TO limit)
        RETURN i WITH RESUME;
    END FOR

    IF backwards = 0 THEN
        RETURN;
    END IF

    FOR i IN (limit TO 1)
        RETURN i WITH RESUME;
    END IF
END PROCEDURE -- sequence
```

SYSTEM

Use the SYSTEM statement to make an operating-system command run from within a procedure.

Syntax



Element	Purpose	Restrictions	Syntax
character variable	A procedure variable that contains a valid operating-system command	The procedure variable must be a character data type variable that is valid in the statement block.	Identifier, p. 1-723
expression	Any expression that is a user-executable operating-system command	You cannot specify that the command run in the background.	Operating-system dependent

Usage

If the supplied expression is not a character expression, *expression* is converted to a character expression before the operating-system command is made. The complete character expression passes to the operating system and executes as an operating-system command.

The operating-system command that the SYSTEM statement specifies cannot run in the background. The database server waits for the operating system to complete execution of the command before it continues to the next procedure statement.

Your procedure cannot use a value or values that the command returns.

If the operating-system command fails (that is, if the operating system returns a nonzero status for the command), an exception is raised that contains the returned operating-system status as the ISAM error code and an appropriate SQL error code.

In DBA- and owner-privileged procedures that contain SYSTEM statements, the operating-system command runs with the permissions of the user who is executing the procedure.

Specifying Environment Variables in SYSTEM Statements

When the operating-system command that SYSTEM specifies is executed, no guarantee exists that the environment variables that the user application set are passed to the operating system. To ensure that the environment variables that the application set are carried forward to the operating system, enter a SYSTEM command that sets the environment variables before you enter the SYSTEM command that causes the operating-system command to execute.

For information on the operating-system commands that set environment variables, see Chapter 4 of the *Informix Guide to SQL: Reference*.

Examples of the SYSTEM Statement in Procedures

The following example shows the use of a SYSTEM statement within a stored procedure. The SYSTEM statement in this example causes the operating system to send a mail message to the system administrator.

```
CREATE PROCEDURE sensitive_update()
    .
    .
    .
    LET mailcall = 'mail headhoncho < alert'
    -- code that evaluates if operator tries to execute a
    -- certain command, then sends email to system
    -- administrator
    SYSTEM mailcall
    .
    .
    .
END PROCEDURE; -- sensitive_update
```

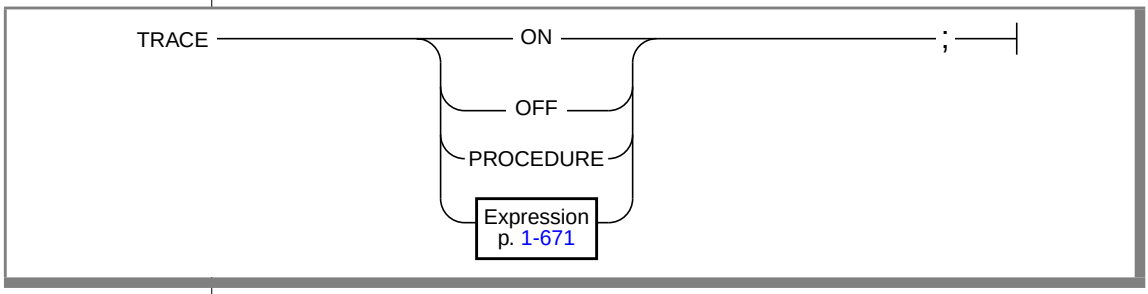
You can use a double-pipe symbol (||) to concatenate expressions with a SYSTEM statement, as the following example shows:

```
CREATE PROCEDURE sensitive_update2()
    DEFINE user1 char(15);
    DEFINE user2 char(15);
    LET user1 = 'joe';
    LET user2 = 'mary';
    .
    .
    .
    -- code that evaluates if operator tries to execute a
    -- certain command, then sends email to system
    -- administrator
    SYSTEM 'mail -s violation' ||user1 || ' ' || user2
                    || '< violation_file';
    .
    .
    .
END PROCEDURE;
```

TRACE

Use the TRACE statement to control the generation of debugging output.

Syntax



Usage

The TRACE statement generates output that is sent to the file that the SET DEBUG FILE TO statement specifies.

Tracing prints the current values of the following items:

- Variables
- Procedure arguments
- Return values
- SQL error codes
- ISAM error codes

The output of each executed TRACE statement appears on a separate line.

If you use the TRACE statement before you specify a DEBUG file to contain the output, an error is generated.

Called procedures inherit the trace state. That is, a called procedure assumes the same trace state (ON, OFF, or PROCEDURE) as the calling procedure. The called procedure can set its own trace state, but that state is not passed back to the calling procedure.

A procedure that is executed on a remote database server does not inherit the trace state.

TRACE ON

If you specify the keyword ON, all statements are traced. The values of variables (in expressions or otherwise) are printed before they are used. To turn tracing ON implies tracing both procedure calls and statements in the body of the procedure.

TRACE OFF

If you specify the keyword OFF, all tracing is turned off.

TRACE PROCEDURE

If you specify the keyword PROCEDURE, only the procedure calls and return values, but not the body of the procedure, are traced.

Printing Expressions

You can use the TRACE statement with a quoted string or an expression to display values or comments in the output file. If the expression is not a literal expression, the expression is evaluated before it is written to the output file.

You can use the TRACE statement with an expression even if you used a TRACE OFF statement earlier in a procedure. However, you must first use the SET DEBUG statement to establish a trace-output file.

The following example uses a TRACE statement with an expression after it uses a TRACE OFF statement:

```
CREATE PROCEDURE tracing ()
    DEFINE i INT;
BEGIN
    ON EXCEPTION IN (1)
    END EXCEPTION; -- do nothing
    TRACE OFF;
    SET DEBUG FILE TO '/tmp/foo.trace';
    TRACE 'Forloop starts';
    FOR i IN (1 TO 1000)
        BEGIN
            TRACE 'FOREACH starts';
```

```
                FOREACH SELECT...INTO a FROM t
                    IF <some condition> THEN
                        RAISE EXCEPTION 1    -- emergency exit
                    END IF
                END FOREACH

                -- return some value
            END
        END FOR

        -- do something
    END;
    END PROCEDURE
```

The following example shows additional TRACE statements:

```
    CREATE PROCEDURE testproc()
        DEFINE i INT;

        TRACE OFF;
        SET DEBUG FILE TO '/tmp/test.trace';
        TRACE 'Entering foo';

        TRACE PROCEDURE;
        LET i = testtoo();

        TRACE ON;
        LET i = i + 1;

        TRACE OFF;
        TRACE 'i+1 = ' || i+1;
        TRACE 'Exiting testproc';

        SET DEBUG FILE TO '/tmp/test2.trace';

    END PROCEDURE;
```

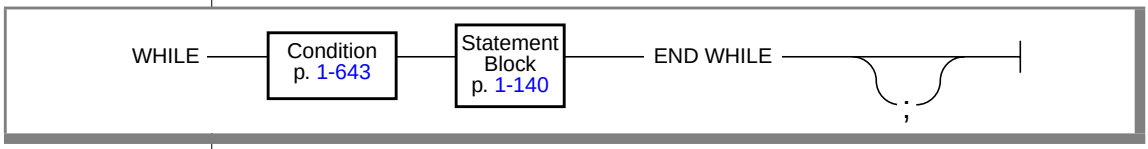
Looking at the Traced Output

To see the traced output, use an editor or utility to display or read the contents of the file.

WHILE

Use the WHILE statement to establish an indefinite loop within a procedure.

Syntax



Usage

The condition is evaluated once at the beginning of the loop, and subsequently at the beginning of each iteration. The statement block is executed as long as the condition remains true. The loop terminates when the condition evaluates to not true.

If any expression within the condition evaluates to null, the condition automatically becomes not true unless you are explicitly testing for the IS NULL condition.

If an expression within the condition has an unknown value because it references uninitialized procedure variables, an immediate error results. In this case, the loop terminates, and an exception is raised.

```
CREATE PROCEDURE simp_while()
    DEFINE i INT;
    DEFINE pf_name CHAR(15);
    WHILE EXISTS (SELECT fname INTO pf_name FROM customer
         WHERE customer_num > 400)
        DELETE FROM customer WHERE id_2 = 2;
    END WHILE

    LET i = 1;
    WHILE i < 10
        INSERT INTO tab_2 VALUES (i);
        LET i = i +1;
    END WHILE;
END PROCEDURE;
```

Index

A

ABS function
  syntax in expression 1-684
  use in expression 1-687
ABSOLUTE keyword
  syntax in FETCH 1-296
  use in FETCH 1-300
Access control. *See* Privilege.
ACCESS FOR keywords, in INFO statement 1-368
ACOS function
  syntax in expression 1-704
  use in expression 1-706
Action clause
  AFTER 1-200
  definition of 1-199
  FOR EACH ROW 1-200
  subset, syntax 1-205
  syntax 1-199
Action statements
  in triggered action clause 1-207
  list of 1-207
  order of execution 1-207
Active set
  constructing with OPEN 1-392, 1-393
  retrieving data with FETCH 1-296
ADD CONSTRAINT keywords, syntax in ALTER TABLE 1-46
AFTER
  action 1-200
  keyword 1-200
Aggregate function
  ALL keyword, syntax 1-709
  AVG function, syntax 1-709
  COUNT function, syntax 1-709
  DISTINCT keyword, syntax 1-709
  in ESQL 1-721
  in EXISTS subquery 1-654
  in expressions 1-464
  in SELECT 1-465
  MAX function, syntax 1-709
  MIN function, syntax 1-709
  RANGE function 1-717
  restrictions with GROUP BY 1-484
  STDEV function 1-718
  SUM function, syntax 1-709
  summary 1-719
  VARIANCE function 1-718
Algebraic functions
  ABS function 1-687
  MOD function 1-687
  POW function 1-688
  ROOT function 1-688
  ROUND function 1-688
  SQRT function 1-689
  syntax 1-684
  TRUNC function 1-690
Algorithm
  for adding columns to tables 1-49
  in-place alter 1-49
Alias
  for a table in SELECT 1-473
  use with GROUP BY clause 1-483, 1-488
ALL keyword
  beginning a subquery 1-479
  DISCONNECT statement 1-264
  syntax
    in expression 1-709

in GRANT 1-349
in REVOKE 1-442
in SELECT 1-461
with UNION operator 1-459
use
in Condition subquery 1-655
in expression 1-711
in GRANT 1-351, 1-352
in REVOKE 1-444
in SELECT 1-463
with UNION operator 1-499
ALLOCATE DESCRIPTOR statement
syntax 1-19
with concatenation operator 1-673
Allocating memory with the ALLOCATE DESCRIPTOR statement 1-19
ALS. *See* Asian Language Support.
ALTER FRAGMENT statement
ADD Clause 1-37
ATTACH clause 1-25
attaching with fragment expression 1-28
attaching with round-robin 1-27
DETACH clause 1-29
DROP clause 1-39
effects on blob columns 1-29
effects on constraints 1-29
effects on indexes 1-29
effects on triggers 1-29
effects on triggers, constraints, indexes 1-26
how executed 1-24
INIT clause 1-30
MODIFY clause 1-40
privileges needed to alter fragments 1-23
syntax 1-22
use 1-23
ALTER INDEX statement
creating clustered index 1-43
dropping clustered index 1-45
syntax 1-43
ALTER keyword
syntax
in GRANT 1-340
in REVOKE 1-442
use
in GRANT 1-350, 1-351
in REVOKE 1-444
Alter privilege 1-350, 1-351
ALTER TABLE statement
ADD clause 1-48
ADD CONSTRAINT clause 1-70
adding
a column 1-48
a column constraint 1-69
a table-level constraint 1-70
rowids 1-75
algorithms for adding columns 1-49
BEFORE option 1-64
cascading deletes 1-61
changing column data type 1-66
changing table lock mode 1-75
CHECK clause 1-63
DEFAULT clause 1-51
DROP clause 1-64
DROP CONSTRAINT clause 1-73
dropping
a column 1-64
a column constraint 1-73
rowids 1-76
LOCK MODE clause 1-75
MODIFY NEXT SIZE clause 1-74
ON DELETE CASCADE keyword 1-57
PAGE keyword 1-75
privilege for 1-340
reclustering a table 1-44
REFERENCES clause 1-57
ROW keyword 1-75
rules for primary key constraints 1-72
rules for unique constraints 1-72
setting columns NOT NULL 1-53
specifying object modes for constraints 1-55
syntax 1-46
American National Standards Institute. *See* ANSI.
AND keyword
syntax in Condition segment 1-643
use
in Condition segment 1-656
with BETWEEN keyword 1-476
AND logical operator 1-656
ANSI compliance
-ansi flag Intro-21, 1-149, 1-155, 1-225
list of SQL statements 1-16
reserved words 1-724
table naming 1-432
ANSI-compliance
updating rows 1-614
ANSI-compliant database
constraint naming 1-658
create with START DATABASE 1-581
description of 1-106
FOR UPDATE not required in 1-238
procedure naming 1-754
synonym naming 1-766
table privileges 1-155
using with INFORMIX-SE 1-108
with BEGIN WORK 1-78
ANY keyword
beginning a subquery 1-479
use in Condition subquery 1-655
Application
comments in 1-10
single-threaded 1-529
thread-safe 1-263, 1-529, 1-532
Arbitrary rule 1-36, 1-126, 1-188
Arithmetic functions. *See* Algebraic functions.
Arithmetic operator, in expression 1-672
Array, moving rows into with FETCH 1-303
AS keyword
in SELECT 1-461
syntax
in CREATE VIEW 1-224
in GRANT 1-340
use
in CREATE VIEW 1-226
in GRANT 1-354
with display labels 1-466
with table aliases 1-474
ASC keyword
syntax

in CREATE INDEX 1-109
in SELECT 1-487
use
in CREATE INDEX 1-113
in SELECT 1-490
ASCII code set 1-763
ASIN function
syntax in expression 1-704
use in expression 1-706
Asterisk (*)
arithmetic operator 1-671
as wildcard character in a condition 1-651
use in SELECT 1-461
ATAN function
syntax in expression 1-704
use in expression 1-706
ATAN2 function
syntax in expression 1-704
use in expression 1-706
Attached index 1-123
Audit trail
applying with RECOVER TABLE 1-425
dropping with DROP AUDIT 1-265
manipulating audit trail file 1-427
no clustered index 1-111
starting with CREATE AUDIT 1-102
Automatic type conversion. *See* Data type conversion.
AVG function
syntax in expression 1-709
use in expression 1-712

B

Backslash (\)
as escape character with LIKE 1-650
as escape character with MATCHES 1-651
Backup. *See* Archiving.
BEFORE keyword
in ALTER TABLE 1-64
in CREATE TRIGGER 1-199
BEGIN WORK statement
locking in a transaction 1-77
syntax 1-77
BETWEEN keyword
syntax in Condition segment 1-644
use
in Condition segment 1-648
in SELECT 1-476
Binary Large Object (BLOB)
attaching tables 1-29
effect of isolation on retrieval 1-559, 1-580
in a LOAD statement 1-384
in an UNLOAD statement 1-607
Boolean expression
in Condition segment 1-643
Btree cleaner list 1-626
BUFFERED keyword, syntax in SET LOG 1-564
BUFFERED LOG keywords
syntax in CREATE DATABASE 1-104
use in CREATE DATABASE 1-107
Buffered logging 1-104
BYTE data type
considerations for UNLOAD statement 1-607
requirements for LOAD statement 1-384
syntax 1-665
with stored procedures 2-10, 2-15

C

Calculated expression, restrictions with GROUP BY 1-484
CALL keyword, in the WHENEVER statement 1-632, 1-638
CALL statement
syntax 2-4
Caret (^) wildcard in Condition segment 1-651
Cascading deletes
defined 1-61, 1-170
locking associated with 1-62, 1-171
logging 1-62, 1-171
restriction 1-62, 1-172
syntax 1-57, 1-165
Cascading triggers
and triggering table 1-213, 1-217
description of 1-216
maximum number of 1-216
scope of correlation names 1-210
triggered actions 1-201
CHAR data type
in INSERT 1-760
syntax 1-665
using as default value 1-52, 1-160
CHARACTER VARYING data type
syntax 1-667
CHARACTER_LENGTH function 1-698
CHAR_LENGTH function 1-698
Check constraint
adding with ALTER TABLE 1-63
definition of 1-172
specifying at column level 1-172
specifying at table level 1-172
CHECK keyword
use in ALTER TABLE 1-63
use in CREATE TABLE 1-172
CHECK TABLE statement, syntax and use 1-79
Checking for corrupted tables 1-79
Client/server
environment 1-219
CLOSE DATABASE statement
prerequisites to close 1-85
syntax 1-85
CLOSE statement
closing a select cursor 1-82
closing an insert cursor 1-82
cursors affected by transaction end 1-84
syntax 1-81
with concatenation operator 1-673
CLUSTER keyword
syntax
in ALTER INDEX 1-43
in CREATE INDEX 1-109
use
in ALTER INDEX 1-44
in CREATE INDEX 1-111

Clustered index
with ALTER INDEX 1-43
with audit trails 1-111
Clustered index, creating with CREATE INDEX 1-111
Collation
with relational operators 1-762
Column
defining as foreign key 1-176
defining as primary key 1-176
displaying information for 1-366
dropping 1-64
effect on constraints 1-65
effect on triggers 1-65
effect on views 1-65
inserting into 1-372
modifying with ALTER TABLE 1-66
naming conventions 1-429
number allowed when defining constraint 1-156
putting a constraint on 1-156
referenced and referencing 1-59, 1-168
renaming 1-428
specifying a subscript 1-489, 1-675
specifying check constraint for 1-172
specifying with CREATE TABLE 1-158
virtual 1-226
Column expression
in SELECT 1-464
syntax 1-673
Column name
using functions as names 1-729
using keywords as names 1-730
when qualified 1-209
Column number
effect on triggers 1-198
Column value
in triggered action 1-211
qualified vs. unqualified 1-211
when unqualified 1-211
Column-level privilege 1-351
COLUMNS FOR keywords, in INFO statement 1-366
Command file
comment symbols in 1-10
defined 1-10
See also Command script.
Command script
See also Command file.
Comment symbol
curly brackets ({}) 1-9
double dash (--) 1-9
examples of 1-10
how to enter 1-9
in application programs 1-10
in command file 1-10
in prepared statements 1-405
in SQL APIs 1-10
in stored procedure 1-10
COMMIT WORK statement
syntax 1-87
use in ANSI-compliant databases 1-88
use in non-ANSI databases 1-87
Committed Read isolation level 1-557
COMMITTED READ keywords, syntax in SET ISOLATION 1-556
Comparison condition
syntax and use 1-644
Complex query
example of 1-552
Composite column list, multiple-column restrictions 1-71, 1-72
Composite index
column limit 1-113
definition of 1-113
Compound assignment 2-32
Concatenation operator (| |) 1-672
Concurrency
Committed Read isolation 1-557
Cursor Stability isolation 1-558
defining with SET ISOLATION 1-556
defining with SET TRANSACTION 1-579
Dirty Read isolation 1-557
Read Committed isolation 1-578
Read Uncommitted isolation 1-578
Repeatable Read isolation 1-558, 1-578
Serializable isolation level 1-578
Condition segment
ALL, ANY, SOME subquery 1-655
boolean expressions 1-644
comparison condition 1-644
description of 1-643
join conditions 1-480
null values 1-644
relational operators in 1-647
subquery in SELECT 1-652
syntax 1-643
use of functions in 1-644
wildcards in searches 1-650
with BETWEEN keyword 1-648
with ESCAPE keyword 1-651
with EXISTS keyword 1-654
with IN keyword 1-648
with IS keyword 1-649
with MATCHES keyword 1-649
with NOT keyword 1-650
CONNECT keyword
in GRANT 1-343
in REVOKE 1-446
Connect privilege 1-343, 1-446
CONNECT statement
and INFORMIXSERVER environment variable 1-91
connection context 1-91
connection identifiers 1-91
database environment 1-94
DEFAULT option 1-91
implicit connections 1-92
syntax 1-89
use 1-90
USER clause 1-99
WITH CONCURRENT TRANSACTION option 1-93
CONNECT TO statement
with concatenation operator 1-673
Connection
context 1-91, 1-262, 1-528
current 1-90, 1-263, 1-532
default 1-92, 1-262, 1-531
dormant 1-90, 1-262, 1-528
identifiers 1-91
implicit 1-92, 1-262, 1-531
Constant expression

in SELECT 1-464
inserting with PUT 1-418
restrictions with GROUP BY 1-484
syntax 1-676
Constraint
adding with ALTER TABLE 1-68, 1-69, 1-70, 1-156, 1-515
affected by dropping a column from table 1-65
checking 1-217
definition of 1-155
dropping with ALTER TABLE 1-73, 1-156
enforcing 1-157
modifying a column that has constraints 1-66
number of columns allowed 1-156, 1-174
object modes for setting with SET 1-501
privileges needed to create 1-72
rules for unique constraints 1-70
specifying at table level 1-174
transaction mode 1-523
with DROP INDEX 1-268
CONSTRAINT keyword
in ALTER TABLE 1-69
in CREATE TABLE 1-162
Consumed table 1-25
CONTINUE keyword, in the WHENEVER statement 1-632, 1-637
CONTINUE statement
syntax 2-7
Conventions
for naming tables 1-154
Correlated subquery
definition of 1-653
Correlation name
and stored procedures 1-210
in COUNT DISTINCT clause 1-210
in GROUP BY clause 1-210
in SET clause 1-210
in stored procedure 1-214
new 1-205
old 1-205
rules for 1-209
scope of 1-210
table of values 1-211
using 1-209
when to use 1-210
COS function
syntax in expression 1-704
use in expression 1-705
COUNT DISTINCT clause 1-210
COUNT field
getting contents with GET DESCRIPTOR 1-314
setting value for WHERE clause 1-542
use in GET DESCRIPTOR 1-316
COUNT function
syntax in expression 1-709
use in expression 1-711, 1-713
COUNT keyword, use in SET DESCRIPTOR 1-543
CREATE AUDIT statement
need for archive 1-103
starts audit trail 1-102
syntax 1-102
CREATE DATABASE statement
ANSI compliance 1-106
logging with OnLine 1-106
syntax 1-104
using with
CREATE SCHEMA 1-147
INFORMIX-SE 1-107
PREPARE 1-105
CREATE INDEX statement
cluster with fragments 1-111
composite indexes 1-113
fragment by expression 1-124
implicit table locks 1-110
IN dbspace clause 1-123
specifying a storage option 1-123
specifying object modes 1-128
syntax 1-109
using with
ASC keyword 1-113
CLUSTER keyword 1-111
CREATE SCHEMA 1-147
DESC keyword 1-113
FILLFACTOR clause 1-122
UNIQUE keyword 1-110
CREATE PROCEDURE FROM statement
syntax and use 1-144
CREATE PROCEDURE statement
syntax 1-134
CREATE ROLE statement 1-145
CREATE SCHEMA statement
defining a trigger 1-193
specifying owner of created objects with GRANT 1-149
syntax 1-147
with CREATE sequences 1-149
CREATE SYNONYM statement
chaining synonyms 1-152
synonym for a table 1-150
synonym for a view 1-150
syntax 1-150
with CREATE SCHEMA 1-147
CREATE TABLE statement
adding rowids 1-183
cascading deletes 1-170
CHECK clause 1-172
creating temporary table 1-177
DEFAULT clause 1-159
defining constraints
at column level 1-162
at table level 1-174
FRAGMENT BY clause 1-186
IN dbspace clause 1-184
LOCK MODE clause 1-190
naming conventions 1-154
ON DELETE CASCADE keywords 1-165
REFERENCES clause 1-165
rules for primary keys 1-167
rules for referential constraints 1-167
rules for unique constraints 1-167
setting columns NOT NULL 1-162
specifying extent size 1-188
specifying table columns 1-158
storing database tables 1-183
syntax 1-154
TEMP TABLE clause 1-177
with BLOB data types 1-163
with CREATE SCHEMA 1-147
WITH ROWIDS clause 1-183
CREATE TRIGGER statement
in ESQL/C 1-193
in ESQL/COBOL 1-193

privilege to use 1-193
purpose 1-192
specifying object modes 1-222
syntax 1-192
triggered action clause 1-206
use 1-193
CREATE VIEW statement
column data types 1-225
privileges 1-225
syntax 1-224
virtual column 1-226
WITH CHECK OPTION 1-227
with CREATE SCHEMA 1-147
with SELECT * notation 1-224
Curly brackets ({}) comment symbol 1-9
Current database
specifying with DATABASE 1-229
CURRENT function
syntax
in Condition segment 1-644
in expression 1-676
in INSERT 1-375
use
in ALTER TABLE 1-51
in CREATE TABLE 1-159
in expression 1-680
in INSERT 1-377
in WHERE condition 1-681
input for DAY function 1-681
CURRENT keyword
in DISCONNECT statement 1-263
in SET CONNECTION statement 1-532
syntax in FETCH 1-296
use in FETCH 1-300
CURRENT OF keywords
syntax
in DELETE 1-252
in UPDATE 1-612
use
in DELETE 1-254
in UPDATE 1-620
Cursor
activating with OPEN 1-390
affected by transaction end 1-84
associating with prepared statements 1-247
characteristics 1-239
closing 1-81
closing with ROLLBACK WORK 1-456
declaring 1-234
definition of types 1-237
for update 1-239, 1-242
restricted statements 1-245
using in ANSI-mode databases 1-246
using in non-ANSI databases 1-246
manipulation statements 1-13
opening 1-392, 1-393
read-only 1-245
defined 1-238
restricted statements 1-245
using in ANSI-mode databases 1-246
using in non-ANSI databases 1-246
retrieving values with FETCH 1-296
scroll 1-240
sequential 1-239
statement, as trigger event 1-195
using with transactions 1-248
with
INTO keyword in SELECT 1-469
prepared statements 1-238
Cursor Stability isolation level 1-558
CURSOR STABILITY keywords, syntax in SET ISOLATION 1-556
Cursory procedure 2-25

D

Data
inserting with the LOAD statement 1-380
Data access statements 1-14
Data definition statements 1-12
Data distributions
confidence 1-629
on temporary tables 1-631
RESOLUTION 1-628, 1-629
DATA field
setting with SET DESCRIPTOR 1-545
Data integrity
statements 1-14
Data manipulation statements 1-13
Data model
See also Relational model.
Data type
changing with ALTER TABLE 1-68
considerations for INSERT 1-376, 1-760
requirements for referential constraints 1-60, 1-169
segment 1-664
specifying with CREATE VIEW 1-225
syntax 1-665
See also each data type listed under its own name.
Database
closing with CLOSE DATABASE 1-85
creating ANSI-compliant 1-581
creating with CREATE DATABASE 1-105
default isolation levels 1-558, 1-579
dropping 1-266
lock 1-231
naming conventions 1-662
naming with variable 1-663
opening in exclusive mode 1-231
optimizing queries 1-626
remote 1-662
renaming 1-431
restoring 1-457
stopping logging on 1-581
Database Administrator (DBA) 1-344
Database Name segment
database outside DBPATH 1-663
for remote database 1-662
naming conventions 1-660
naming with variable 1-663

syntax 1-660
using quotes, slashes 1-663
Database object
meaning in SET statement 1-502
DATABASE statement
determining database type 1-229
exclusive mode 1-231
specifying current database 1-229
SQLWARN after 1-230
syntax 1-229
using with program variables 1-231
Database-level privilege
description of 1-342
granting 1-342
passing grant ability 1-353
revoking 1-446
See also Privilege.
Dataskip
skipping unavailable dbspaces 1-534
DATE data type
functions in 1-699
syntax 1-665
DATE function
syntax in expression 1-699
use in expression 1-700
DATETIME data type
as quoted string 1-759
field qualifiers 1-669
in
expression 1-681
INSERT 1-760
syntax 1-665, 1-746
DATETIME Field Qualifier segment 1-669
DAY function
syntax in expression 1-699
use
in expression 1-701
DAY keyword
syntax
in DATETIME data type 1-669
in INTERVAL data type 1-743
use
as DATETIME field qualifier 1-746
as INTERVAL field qualifier 1-749
DBA keyword in REVOKE 1-446
DBANSIWARN environment variable 1-149, 1-155, 1-225
DBA-privileged procedure 1-135
DBCENTURY environment variable 1-383
DBDATE environment variable 1-383, 1-606
DBDELIMITER environment variable 1-385
DBINFO function
syntax in expression 1-690
use in expression 1-690
DBMONEY environment variable 1-383, 1-606
DBMS. *See* Database management system.
DBPATH environment variable 1-231, 1-663
.dbs extension 1-105, 1-179, 1-231
DBSERVERNAME function
returning server name 1-679
use
in ALTER TABLE 1-51
in CREATE TABLE 1-159
in expression 1-679
dbspace
skipping if unavailable 1-534
DBSPACETEMP environment variable
for locating temporary tables 1-178
DBTIME environment variable 1-383, 1-607
DDL statements, summary 1-12
Deadlock detection 1-562
DEALLOCATE DESCRIPTOR statement
syntax 1-232
with concatenation operator 1-673
DECIMAL data type
syntax 1-665
using as default value 1-52, 1-160
DECLARE statement
cursor characteristics 1-239
cursor types 1-237
cursors with prepared statements 1-247
cursors with transactions 1-248
definition and use
hold cursor 1-240
insert cursor 1-239, 1-250
procedure cursor 1-238
read-only cursor 1-238, 1-245
scroll cursor 1-240
select cursor 1-238
sequential cursor 1-239
update cursor 1-239, 1-242
FOR READ ONLY 1-238, 1-245
insert cursor 1-237
insert cursor with hold 1-251
procedure cursor 1-237
restrictions with SELECT with ORDER BY 1-491
syntax 1-234
update cursor 1-237
updating specified columns 1-243
use
with concatenation operator 1-673
with FOR UPDATE keywords 1-239
with SELECT 1-470
DEFAULT keyword
in CONNECT statement 1-91
in DISCONNECT statement 1-262
in SET CONNECTION statement 1-531
Default value
specifying
with ALTER TABLE 1-52
with CREATE TABLE 1-159
DEFINE statement
placement of 2-9
syntax 2-8
DELETE keyword
syntax
in GRANT 1-349
in REVOKE 1-442
use
in GRANT 1-350
in REVOKE 1-443
Delete privilege 1-349
DELETE REFERENCING clause

and FOR EACH ROW section 1-205
syntax 1-203
DELETE statement
as triggering statement 1-195
cascading 1-253
CURRENT OF clause 1-254
in trigger event 1-194
in triggered action 1-207
privilege for 1-349
syntax 1-252
with cursor 1-242
with SELECT...FOR UPDATE 1-492
within a transaction 1-252
DELIMIDENT environment variable 1-728
Delimited identifers
non-ASCII characters in 1-728
Delimited identifier
defined 1-727
multibyte characters in 1-728
syntax 1-726
Delimiter
for LOAD input file 1-385
specifying with UNLOAD 1-608
DELIMITER keyword
in LOAD 1-385
in UNLOAD 1-608
Demonstration database
copying Intro-9
installation script Intro-8
overview Intro-8
See also stores7 database.
DESC keyword 1-490
syntax
in CREATE INDEX 1-109
in SELECT 1-487
use
in CREATE INDEX 1-113
in SELECT 1-490
DESCRIBE statement
and the USING SQL DESCRIPTOR clause 1-258
describing statement type 1-256
INTO sqlda pointer clause 1-259
relation to GET DESCRIPTOR 1-317
syntax 1-255
using with concatenation operator 1-673
values returned by SELECT 1-257
Descriptor 1-258
Detached index 1-123
Diagnostics table
creating with START VIOLATIONS TABLE 1-584
examples 1-510, 1-516, 1-599, 1-601, 1-604
how to start 1-506, 1-584
how to stop 1-507, 1-603
privileges on 1-597
relationship to target table 1-588
relationship to violations table 1-588
restriction on dropping 1-275
structure 1-596
use with SET 1-505
when to start 1-506
Directory, extension, .dbs 1-105, 1-231
Dirty Read isolation level 1-557
DIRTY READ keywords, syntax in SET ISOLATION 1-556
Disabled object mode
benefits of 1-521
defined 1-509
DISCONNECT statement
ALL keyword 1-264
CURRENT keyword 1-263
DEFAULT option 1-262
with concatenation operator 1-673
Display label
syntax in SELECT 1-461
DISTINCT keyword
syntax
in CREATE INDEX 1-109
in expression 1-709
in SELECT 1-461
use
in CREATE INDEX 1-110
in SELECT 1-463
no effect in subquery 1-654
Distributions
dropping with DROP DISTRIBUTIONS clause 1-628
privileges required to create 1-629
using the HIGH keyword 1-629
using the MEDIUM keyword 1-629
Division (/) symbol, arithmetic operator 1-671
DML statements, summary 1-13
Documentation notes Intro-28
Double-dash (--) comment symbol 1-9
DROP AUDIT statement 1-265
DROP CONSTRAINT keywords
syntax in ALTER TABLE 1-46
use in ALTER TABLE 1-73
DROP DATABASE statement 1-266
DROP INDEX statement
syntax 1-268
DROP keyword
syntax in ALTER TABLE 1-46
use in ALTER TABLE 1-64
DROP ROLE statement 1-271
DROP SYNONYM statement 1-272
DROP TABLE statement 1-274
DROP TRIGGER statement
syntax 1-277
use of 1-277
DROP VIEW statement 1-279
Duplicate values
in a query 1-463
Dynamic management statements 1-13

E

Enabled object mode
benefits of 1-522
defined 1-509
Environment variable
setting
with SYSTEM statement 2-44
ERROR 1-637
Error checking
continuing after error in stored procedure 2-38
error status with ON EXCEPTION 2-35

with SYSTEM 2-43
ERROR keyword, in the WHENEVER statement 1-632, 1-637
Errors
detecting with WHENEVER 1-635
ESCAPE keyword
syntax in Condition segment 1-644
use
in Condition segment 1-649
with LIKE keyword 1-477, 1-651
with MATCHES keyword 1-478, 1-652
with WHERE keyword 1-477
EXCLUSIVE keyword
syntax
in DATABASE 1-229
in LOCK TABLE 1-387
use
in DATABASE 1-231
in LOCK TABLE 1-388
EXECUTE IMMEDIATE statement
restricted statement types 1-291
syntax and usage 1-290
using with concatenation operator 1-673
EXECUTE ON keywords
syntax in GRANT 1-340
use
in GRANT 1-345
EXECUTE PROCEDURE statement 1-293
associating cursor with DECLARE 1-238
in FOREACH 2-23
in triggered action 1-207
with
INTO keyword 1-302
EXECUTE statement
and sqlca record 1-283
error conditions with 1-289
parameterizing a statement 1-287
syntax 1-281
USING DESCRIPTOR clause 1-289
with concatenation operator 1-673
with INTO keyword 1-283
with USING keyword 1-286
EXISTS keyword
beginning a subquery 1-478
use in condition subquery 1-654
EXIT statement
syntax 2-16
EXP function
syntax in expression 1-695
use in expression 1-695
Explicit temporary table 1-177
Exponential function
EXP function 1-695
Expression
in UPDATE 1-618
ordering by 1-491
Expression segment
aggregate expressions 1-709
column expressions 1-673
combined expressions 1-721
constant expressions 1-676
expression types 1-671
function expressions 1-683
in SPL expressions 1-722
syntax 1-671
Expression-based distribution scheme
in ALTER FRAGMENT 1-35
in CREATE INDEX 1-124
in CREATE TABLE 1-187
EXTEND function
syntax in expression 1-699
use in expression 1-702
Extension, to SQL
symbol for Intro-21
extension, .lok 1-562
Extent
modifying size of new extents with ALTER TABLE 1-74
EXTENT SIZE keywords 1-188

F

FETCH statement
as affected by CLOSE 1-82
checking results with SQLCA 1-306
fetching a row for update 1-305
locking for update 1-305
relation to GET DESCRIPTOR 1-314
specifying memory location of a value 1-301
syntax 1-296
with
concatenation operator 1-673
program arrays 1-303
scroll cursor 1-300
sequential cursor 1-299
X/Open mode 1-299
Field qualifier
for DATETIME 1-669
for INTERVAL 1-743, 1-749
File
extension
.lok 1-562
sending output with the OUTPUT statement 1-400
FILLFACTOR clause in CREATE INDEX 1-122
Filtering object mode
benefits of 1-523
defined 1-510
finding location of row 1-675
FIRST keyword
syntax in FETCH 1-296
use in FETCH 1-300
FLOAT data type
syntax 1-665
using as default value 1-52, 1-160
FLUSH statement
syntax 1-308
with concatenation operator 1-673
FOR EACH ROW action
SELECT statement in 1-201
triggered action list 1-200
FOR keyword
in CONTINUE 2-7
in CREATE AUDIT 1-102
in CREATE SYNONYM 1-150
in EXIT 2-16
FOR READ ONLY keywords
use

in DECLARE 1-245
FOR statement
specifying multiple ranges 2-21
syntax 2-18
using expression lists 2-21
using increments 2-20
FOR TABLE keywords, in UPDATE STATISTICS 1-624
FOR UPDATE keywords
relation to UPDATE 1-621
syntax
in DECLARE 1-234
in SELECT 1-459
use
in DECLARE 1-239, 1-242, 1-248
in SELECT 1-492
with column list 1-243
FOREACH keyword
in CONTINUE statement 2-7
in EXIT 2-16
FOREACH statement
syntax 2-23
Foreign key 1-60, 1-168, 1-176
FOREIGN KEY keywords
in ALTER TABLE 1-69
in CREATE TABLE 1-174
FRACTION keyword
syntax
in DATETIME data type 1-669
in INTERVAL data type 1-743
use
as DATETIME field qualifier 1-746
as INTERVAL field qualifier 1-749
Fragment
modifying with ALTER FRAGMENT 1-22
Fragmentation
adding a fragment 1-37
adding rowids with ALTER TABLE 1-75
adding rowids with CREATE TABLE 1-183
altering fragments 1-22
alter, attaching tables 1-25
and BLOB columns 1-29
combining tables 1-25
defining a new fragmentation strategy 1-25
defining and initializing 1-30
detaching a table fragment 1-29
dropping an existing fragment 1-39
dropping rowids with ALTER TABLE 1-76
expression-based distribution scheme 1-35, 1-124, 1-187
fragment expressions 1-28, 1-35, 1-126
if you run out of log / disk space 1-24
indexes, by expression with CREATE INDEX 1-124
modifying an existing fragment expression 1-40
modifying indexes 1-42
number of rows in fragment 1-24
reinitializing fragmentation strategy 1-30
reinitializing strategy with INIT clause 1-32
remainder 1-39
round robin 1-35
round-robin distribution scheme 1-187
rowid 1-35
rules 1-187
setting priority levels for PDQ 1-568
skipping an unavailable dbspace 1-534
tables, syntax 1-186
Fragment-level privilege
granting with GRANT FRAGMENT 1-356
revoking with REVOKE FRAGMENT 1-450
FREE statement
effect on cursors 1-399
syntax 1-311
with concatenation operator 1-673
FROM keyword
syntax
in PUT 1-416
in SELECT 1-459
use
in PUT 1-420
in SELECT 1-472
Function
algebraic 1-684
Function expression
Algebraic functions 1-683
CHARACTER_LENGTH 1-698
CHAR_LENGTH 1-698
DBINFO function 1-690
description of 1-683
Exponential and logarithmic functions 1-683
HEX function 1-683
in SELECT 1-464
LENGTH function 1-683, 1-697
OCTET_LENGTH 1-698
Time functions 1-683
Trigonometric functions 1-683
TRIM function 1-683

G

GET DESCRIPTOR statement
syntax 1-314
the COUNT keyword 1-316
use with FETCH statement 1-304
with concatenation operator 1-673
X / Open mode 1-317
GET DIAGNOSTICS statement
CLASS_ORIGIN keyword 1-331
CONNECTION_NAME keyword 1-334
exception clause 1-329
MESSAGE_LENGTH keyword 1-331
MESSAGE_TEXT keyword 1-331
MORE keyword 1-328
NUMBER keyword 1-328
purpose 1-321
RETURNED_SQLSTATE keyword 1-331
ROW_COUNT keyword 1-328
SERVER_NAME keyword 1-331
SQLSTATE codes 1-323
statement clause 1-327
keywords 1-327

SUBCLASS_ORIGIN
keyword 1-331
syntax 1-321
Global Language Support (GLS)
default locale Intro-7
nondefault locales Intro-7
GLS. *See* Global Language Support.
GL_DATE environment
variable 1-383, 1-606
GL_DATETIME environment
variable 1-383, 1-607
GO TO keywords, in the
WHENEVER statement 1-632, 1-637
GOTO keyword, in the
WHENEVER statement 1-632, 1-637
GRANT FRAGMENT statement
AS grantor clause 1-363
syntax 1-356
usage 1-356
WITH GRANT OPTION
clause 1-362
GRANT statement
changing grantor 1-354
creating a privilege chain 1-353
database-level privileges 1-342
default table privileges 1-352
passing grant ability 1-353
privileges on a view 1-355
role privileges 1-345
syntax 1-340
table-level privileges 1-350
ALL keyword 1-352
ALTER keyword 1-350
with CREATE SCHEMA 1-147
GROUP BY clause 1-210
GROUP BY keywords
syntax in SELECT 1-459
use in SELECT 1-483

H

Hash rule 1-36, 1-126, 1-188
HAVING keyword
syntax in SELECT 1-459
use in SELECT 1-485
HEX function 1-675
syntax in expression 1-696
use in expression 1-696
HIGH keyword 1-566
Hold cursor
definition of 1-239
insert cursor with hold 1-251
use of 1-240
HOUR keyword
syntax
in DATETIME data type 1-669
in INTERVAL data type 1-743
use
as DATETIME field
qualifier 1-746
as INTERVAL field
qualifier 1-749

I

IDATA field
SET DESCRIPTOR
statement 1-545
with X/Open programs 1-317
Identifier segment
column names 1-734
cursor name 1-736, 1-739
delimited identifiers 1-726
multibyte characters in 1-726
non-ASCII characters in 1-726
stored procedures 1-736
syntax 1-723
table names 1-733, 1-734
using keywords as column
names 1-730
variable name 1-738
IF statement
syntax 2-27
syntax and use 2-27
with null values 2-28
ILENGTH field
SET DESCRIPTOR
statement 1-545
with X/Open programs 1-317
Implicit temporary table 1-178
IN keyword
syntax
in CREATE AUDIT 1-102
in CREATE DATABASE 1-104
in CREATE TABLE 1-183
in LOCK TABLE 1-387
use
in Condition segment 1-648
in Condition subquery 1-653
with WHERE keyword 1-476
Index
attached, defined 1-123
bidirectional traversal 1-114
cleaner list. *See* Btree cleaner list.
clustered fragments 1-111
converting during
upgrade 1-623, 1-627
creating fragments 1-124
creating with CREATE
INDEX 1-109
detached 1-123
displaying information for 1-367
dropping with DROP
INDEX 1-268
duplicate 1-110
effected with ALTER
FRAGMENT ATTACH 1-26
effects on altered fragments 1-29
naming conventions 1-659, 1-742
on ORDER BY columns 1-114, 1-492
provide for expansion 1-122
sharing with constraints 1-157
surviving 1-22
unique
adding when duplicate values
exist 1-130, 1-515
restrictions 1-33
with temporary tables 1-497
with unique and referential
constraints 1-110
index
number allowed on same
columns 1-120
INDEX keyword
syntax
in GRANT 1-349
in REVOKE 1-442
use
in GRANT 1-351
in REVOKE 1-443
Index Name segment
syntax 1-658, 1-741

use 1-742
Index privilege 1-351
INDEXES FOR keywords, in INFO statement 1-367
INDICATOR field
setting with SET DESCRIPTOR 1-547
INDICATOR keyword, in SELECT 1-469, 1-470, 1-471
Indicator variable
in expression 1-721
in SELECT 1-469, 1-470, 1-471
INFO statement
displaying privileges and status 1-368
displaying tables, columns, and indexes 1-366
syntax 1-365
INFORMIX-OnLine Dynamic Server
and triggering statement 1-194
INFORMIX-OnLine/Optical
list of statements 1-15
INFORMIXSERVER environment variable 1-91
informix, privileges associated with user 1-344
In-place alter algorithm 1-49
Insert buffer
counting inserted rows 1-310, 1-423
filling with constant values 1-418
inserting rows with a cursor 1-373
storing rows with PUT 1-417
triggering flushing 1-422
Insert cursor
closing 1-82
definition of 1-237
in INSERT 1-373
in PUT 1-419
opening 1-394
reopening 1-396
result of CLOSE in SQLCA 1-82
use of 1-239
with hold 1-251
INSERT INTO keywords
in INSERT 1-370
in LOAD 1-386
INSERT keyword
syntax
in GRANT 1-349
in REVOKE 1-442
use
in GRANT 1-350
in REVOKE 1-443
INSERT REFERENCING clause
and FOR EACH ROW section 1-205
syntax 1-202
INSERT statement
effect of transactions 1-374
filling insert buffer with PUT 1-417
in dynamic SQL 1-379
in trigger event 1-194
in triggered action 1-207
inserting
nulls with the VALUES clause 1-378
rows through a view 1-372
rows with a cursor 1-373
SERIAL columns 1-377
specifying values to insert 1-375
syntax 1-370
use with insert cursor 1-250
using functions in the VALUES clause 1-377
with
DECLARE 1-234
SELECT 1-378
INTEGER data type
syntax 1-665
using as default value 1-52, 1-160
Integrity. *See* Data integrity.
INTERVAL data type
as quoted string 1-759
field qualifier, syntax 1-743
in expression 1-682
in INSERT 1-760
syntax 1-665, 1-749
INTERVAL Field Qualifier segment 1-743
INTO keyword
in SELECT 1-467
syntax
in FETCH 1-296
in SELECT 1-459
use
in FETCH 1-303
in SELECT 1-467
INTO TEMP keywords
syntax in SELECT 1-459
use
in SELECT 1-495
with UNION operator 1-499
IS keyword
in Condition segment 1-649
with WHERE keyword 1-476
IS NOT keywords, syntax in Condition segment 1-644
IS NULL keywords 1-476
ISAM error code 2-34, 2-39
Isolation level
Committed Read 1-557
Cursor Stability 1-558
definitions 1-557, 1-578
Dirty Read 1-557
in external tables 1-559, 1-575
Read Committed 1-578
Read Uncommitted 1-578
Repeatable Read 1-558, 1-578
Serializable 1-578
use with FETCH statement 1-305
ITYPE field
SET DESCRIPTOR statement 1-545
setting with SET DESCRIPTOR 1-547
with X/Open programs 1-317

J

Join
in Condition segment 1-480
multiple-table join 1-481
outer join 1-482
self-join 1-482
two-table join 1-481
Join column. *See* Foreign key.

K

Keywords
SQL 1-724
using in triggered action 1-208

L

LAST keyword
  syntax in FETCH 1-296
  use in FETCH 1-300
LENGTH field
  setting with SET DESCRIPTOR 1-545
  with DATETIME and INTERVAL types 1-546
  with DECIMAL and MONEY types 1-546
LENGTH function
  in expression 1-464
  syntax in expression 1-683, 1-697
  use in expression 1-698
LET statement
  syntax 2-31
LIKE keyword
  syntax in Condition segment 1-644
  use in SELECT 1-477
  wildcard characters 1-477
List-mode format, in SET 1-502
Literal
  DATETIME
    in Condition segment 1-644
    in expression 1-676, 1-681
    segment 1-746
    syntax 1-746
    syntax in INSERT 1-375
    use in ALTER TABLE 1-51
    use in CREATE TABLE 1-159
    with IN keyword 1-476
  DATE, using as a default value 1-52, 1-160
  INTERVAL
    in Condition segment 1-644
    in expression 1-676, 1-682
    segment 1-749
    syntax 1-749
    syntax in INSERT 1-375
    using as default value 1-52, 1-160
  Number
    in Condition segment 1-644
    in expression 1-676, 1-680
    segment 1-752
    syntax 1-752
    syntax in INSERT 1-375
    with IN keyword 1-649
LOAD statement
  DELIMITER clause 1-385
  INSERT INTO clause 1-386
  loading VARCHAR, TEXT, or BYTE data types 1-384
  specifying the table to load into 1-386
  syntax 1-380
  the LOAD FROM file 1-382
Locale
  default Intro-7
  non-default Intro-7
LOCK MODE keywords
  syntax
    in ALTER TABLE 1-46
    in CREATE TABLE 1-190
  use
    in ALTER TABLE 1-75
    in CREATE TABLE 1-190
LOCK TABLE statement
  in databases with transactions 1-388
  in databases without transactions 1-389
  syntax 1-387
Locking
  during
    delete 1-252
    inserts 1-374
    updates 1-243, 1-615
  overriding row-level 1-388
  releasing with COMMIT WORK 1-87
  releasing with ROLLBACK WORK 1-455
  types of locks
    page lock 1-190
    row lock 1-190
  update cursors effect on 1-243
  update locks 1-615
  waiting period 1-562
  with
    FETCH 1-305
    SET ISOLATION 1-556
    SET LOCK MODE 1-561
    SET TRANSACTION 1-575
    UNLOCK TABLE 1-610
  within transaction 1-77
LOG IN keywords, syntax in CREATE DATABASE 1-104
LOG10 function
  syntax in expression 1-696
  use in expression 1-696
Logarithmic function
  syntax
    LOG10 function 1-695
    LOGN function 1-695
  use
    LOG10 function 1-696
    LOGN function 1-695
Logging
  buffered versus unbuffered 1-564
  cascading deletes 1-62, 1-171
  changing mode with SET LOG 1-564
  finding log-file location 1-108
  renaming log 1-582
  setting with CREATE TABLE 1-182
  starting with START DATABASE 1-108, 1-581
  stopping 1-582
  stopping with START DATABASE 1-581
  with INFORMIX-OnLine 1-106
  with INFORMIX-SE 1-107
  with triggers 1-220
Logical operator
  in Condition segment 1-656
LOGN function
  syntax in expression 1-695
  use in expression 1-695
.lok extension 1-562
Loop
  controlled 2-18
  indefinite with WHILE 2-49
LOW keyword 1-566

M

Machine notes Intro-28
Mail, sending from stored procedure 2-44
MATCHES keyword

syntax in Condition segment 1-644
use
in Condition segment 1-649
in SELECT 1-477
wildcard characters 1-478
MAX function
syntax in expression 1-709
use in expression 1-712
MDY function
syntax in expression 1-699
Memory
allocating for a system sqlda structure 1-19
MIN function
syntax in expression 1-709
use in expression 1-713
Minus (-) sign, arithmetic operator 1-671
MINUTE keyword
syntax
in DATETIME data type 1-669
in INTERVAL data type 1-743
use
as DATETIME field qualifier 1-746
as INTERVAL field qualifier 1-749
MOD function
syntax in expression 1-684
use in expression 1-687
MODE ANSI keywords
syntax
in CREATE DATABASE 1-104
in START DATABASE 1-581
use
in CREATE DATABASE 1-107
in START DATABASE 1-582
Model. *See* Data model.
MODIFY keyword
syntax in ALTER TABLE 1-46
use in ALTER TABLE 1-66
MODIFY NEXT SIZE keywords
syntax in ALTER TABLE 1-46
use in ALTER TABLE 1-74
MONEY data type
syntax 1-665
using as default value 1-52
MONEY data type, using as default value 1-160
MONTH function
syntax in expression 1-699
use in expression 1-701
MONTH keyword
syntax
in DATETIME data type 1-669
in INTERVAL data type 1-743
use
as DATETIME field qualifier 1-746
as INTERVAL field qualifier 1-749
MS-DOS operating system. *See* DOS operating system.
Multiple triggers
column numbers in 1-198
example 1-197
order of execution 1-198
preventing overriding 1-219
Multiplication sign, arithmetic operator 1-671
Multirow query
destination of returned values 1-301
managing with FETCH 1-298

N

Naming convention
column 1-429
database 1-662
index 1-659, 1-742, 1-767
table 1-154, 1-770
NCHAR data type
syntax 1-667
Nested ordering, in SELECT 1-490
NEW keyword
in DELETE REFERENCING clause 1-204
in INSERT REFERENCING clause 1-202
in UPDATE REFERENCING clause 1-205
NEXT keyword
syntax in FETCH 1-296
use in FETCH 1-300
NEXT SIZE keywords, use in CREATE TABLE 1-188
NODEFDAC environment variable
effects on new stored procedure 1-135
NOT CLUSTER keywords
syntax in ALTER INDEX 1-43
use in ALTER TABLE 1-45
NOT FOUND keywords in WHENEVER statement 1-632
NOT IN keywords, use in Condition subquery 1-653
NOT keyword
syntax
in Condition segment 1-643, 1-644
with BETWEEN keyword 1-476
with IN keyword 1-478
use
in Condition segment 1-650
with LIKE, MATCHES keywords 1-477
NOT NULL keywords
syntax
in ALTER TABLE 1-54
in CREATE TABLE 1-162
use
in ALTER TABLE 1-53
in CREATE TABLE 1-162
with IS keyword 1-476
NOT operator, condition 1-644
NOT WAIT keywords in SET LOCK MODE 1-561
NULL keyword, ambiguous as procedure variable 1-737
Null value
checking for in SELECT 1-284, 1-287
in SPL IF statement 2-28
inserting with the VALUES clause 1-378
returned implicitly by stored procedure 2-41
specifying as default value 1-53
updating a column 1-618
used in Condition with NOT operator 1-644
used in the ORDER BY clause 1-490

with SPL WHILE statement 2-49
NVARCHAR data type
syntax 1-667

O

Object database in SET statement 1-502
Object mode
disabled
benefits of 1-521
defined 1-509
enabled
benefits of 1-522
defined 1-509
examples 1-510, 1-516
filtering
benefits of 1-523
defined 1-510
error options in SET 1-504
for triggers 1-222
meaning in SET statement 1-502
privileges required for changing 1-502
setting with SET 1-501
use
with data definition statements 1-515
with data manipulation statements 1-509
OCTET_LENGTH function 1-698
OF keyword
syntax in DECLARE 1-234
use in DECLARE 1-243
OLD keyword
in DELETE REFERENCING clause 1-204
in INSERT REFERENCING clause 1-202
in UPDATE REFERENCING clause 1-205
ON DELETE CASCADE keyword,DELETE trigger event 1-194
ON EXCEPTION statement
placement of 2-35
syntax 2-34
ON keyword
syntax
in CREATE INDEX 1-109
in GRANT 1-340
use
in CREATE INDEX 1-110
in GRANT 1-352
On-line files Intro-28
OPEN statement
constructing the active set 1-392
opening a procedure cursor 1-393
opening an insert cursor 1-394
opening select or update cursors 1-392
reopening a cursor 1-395, 1-396
substituting values for ? parameters 1-396
syntax 1-390
with concatenation operator 1-673
with FREE 1-399
Optimization, specifying a high or low level 1-566
Optimizer
and SET OPTIMIZATION statement 1-566
with UPDATE STATISTICS 1-626, 1-631
Optimizing
a query 1-548
a server 1-566
across a network 1-567
OR keyword
syntax in Condition segment 1-643
use in Condition segment 1-656
ORDER BY keywords
ascending order 1-490
descending order 1-490
indexes on ORDER BY columns 1-114, 1-492
restrictions in INSERT 1-378
select columns by number 1-491
syntax in SELECT 1-459
use
in SELECT 1-487
with UNION operator 1-498
Order of execution, of action statements 1-207
Outer join, forming 1-473
OUTER keyword, with FROM keyword in SELECT 1-472
OUTPUT statement, syntax and use 1-400
Owner
in ALTER TABLE 1-47
in CREATE SYNONYM 1-150
in Index Name segment 1-658, 1-741
in Procedure Name segment 1-754
in RENAME COLUMN 1-429
in RENAME TABLE 1-432
in Synonym Name segment 1-766
in Table Name segment 1-768
in View Name segment 1-772
of view in CREATE VIEW 1-773
Owner-privileged procedure 1-135

P

PAGE keyword
use in ALTER TABLE 1-75
use in CREATE TABLE 1-190
Parallel distributed queries
SET PRIORITY statement 1-568
Parameter
BYTE or TEXT in SPL 2-15
in CALL statement 2-5
Parameterizing
defined 1-287
prepared statements 1-287
Parameterizing a statement
with SQL identifiers 1-409
Parent-child relationship 1-59, 1-168
PDQ
SET PDQPRIORITY statement 1-568
PDQPRIORITY, SET PDQPRIORITY statement 1-568
Percent (%) sign, wildcard in Condition segment 1-650
PERFORM keyword, in the WHENEVER statement 1-632, 1-639
Permission, with SYSTEM 2-44
Permission. *See* Privilege.

Phantom row 1-557, 1-578
PIPE keyword, in the OUTPUT statement 1-401
Plus (+) sign, arithmetic operator 1-671
POW function
  syntax in expression 1-684
  use in expression 1-688
PRECISION field
  with GET DESCRIPTOR 1-318
  with SET DESCRIPTOR 1-546
PREPARE statement
  executing 1-281
  increasing performance efficiency 1-414
  multi-statement text 1-412, 1-414
  parameterizing a statement 1-408
  parameterizing for SQL identifiers 1-409
  question (?) mark as placeholder 1-403
  releasing resources with FREE 1-313
  restrictions with SELECT 1-405
  statement identifier use 1-403
  syntax 1-402
  valid statement text 1-405
  with concatenation operator 1-673
Prepared statement
  comment symbols in 1-405
  describing returned values with DESCRIBE 1-256
  executing 1-281
  parameterizing 1-287
  prepared object limit 1-403
  valid statement text 1-405
PREVIOUS keyword
  syntax in FETCH 1-296
  use in FETCH 1-300
Primary key constraint 1-59
  data type conversion 1-68
  defining column as 1-176
  dropping 1-74
  enforcing 1-157
  modifying a column with 1-67
  referencing 1-60
  requirements for 1-54, 1-176
  rules of use 1-72, 1-168
PRIMARY KEY keywords
  in ALTER TABLE 1-69
  in CREATE TABLE 1-162, 1-174
PRIOR keyword
  syntax in FETCH 1-296
  use in FETCH 1-300
Privilege
  Alter 1-351
  Connect 1-343
  DBA 1-344
  default for table using CREATE TABLE 1-155
  Delete 1-350
  displaying with the INFO statement 1-368
  for triggered action 1-215
  fragment-level
    defined 1-358
    duration of 1-360
    granting with GRANT FRAGMENT 1-356
    revoking with REVOKE FRAGMENT 1-450
    role in command validation 1-359
  granting to roles 1-345
  granting with GRANT 1-340
  Index 1-351
  Insert 1-350
  needed
    to create a view 1-355
    to drop an index 1-268
    to modify data 1-350
  on a synonym 1-150
  on a table fragment 1-356, 1-450
  on a view 1-225
  Resource 1-343
  revoking with REVOKE 1-437
  Update 1-350
  when privileges conflict 1-341
PRIVILEGES FOR keywords, in INFO statement 1-368
PRIVILEGES keyword
  syntax
    in GRANT 1-349
    in REVOKE 1-442
  use
    in GRANT 1-351
    in REVOKE 1-444
Procedure cursor
  in DECLARE statement 1-237
  opening 1-393
  reopening 1-395
Procedure name
  conflict with function name 1-755
  naming conventions 1-755
Procedure Name segment
  syntax 1-754
Promotable lock 1-243
PUBLIC keyword
  syntax
    in GRANT 1-340
    in REVOKE 1-443
  use
    in GRANT 1-344
    in REVOKE 1-443
PUT statement
  FLUSH with 1-417
  impact on trigger 1-195
  source of row values 1-418
  syntax 1-416
  use in transactions 1-417
  with concatenation operator 1-673

Q

Qualifier, field
  for DATETIME 1-669, 1-746
  for INTERVAL 1-743, 1-749
Query
  optimization information statements 1-14
  piping results to another program 1-401
  sending results to an operating system file 1-400
  sending results to another program 1-401
Question mark (?)
  as placeholder in PREPARE 1-403
  naming variables in PUT 1-420
  replacing with USING keyword 1-396
  wildcard in Condition segment 1-651
Quotation marks

double
around delimited identifier 1-726
around quoted string 1-757
within delimited identifier 1-729
within quoted string 1-759
single
around quoted string 1-757
within quoted string 1-759
Quoted string
in expression 1-676
syntax
in Condition segment 1-644
in expression 1-709
in INSERT 1-375
use
in expression 1-678
in INSERT 1-760
with LIKE, MATCHES keywords 1-477
Quoted String segment
DATETIME, INTERVAL values as strings 1-759
syntax 1-757
wildcards 1-759
with LIKE in a condition 1-759

R

RAISE EXCEPTION statement
syntax 2-39
RANGE function 1-717
Range rule 1-35, 1-126, 1-187
Read Committed isolation level 1-578
READ COMMITTED keywords, syntax in SET TRANSACTION 1-575
Read Uncommitted isolation level 1-578
READ UNCOMMITTED keywords, syntax in SET TRANSACTION 1-575
RECOVER TABLE statement
archiving a database with audit trails 1-425
manipulating audit trail file 1-427
syntax 1-425
REFERENCES FOR keywords, in INFO statement 1-368
REFERENCES keyword
in ALTER TABLE 1-57
in CREATE TABLE 1-165, 1-170
syntax
in GRANT 1-349
in REVOKE 1-442
use
in GRANT 1-351
in REVOKE 1-444
References privilege
definition of 1-351
displaying with the INFO statement 1-368
REFERENCING clause
DELETE REFERENCING clause 1-203
INSERT REFERENCING clause 1-202
UPDATE REFERENCING clause 1-204
using referencing 1-209
Referential constraint
and a DELETE trigger 1-194
data type restrictions 1-169
definition of 1-59, 1-167
dropping 1-74
enforcing 1-157
modifying a column with 1-67
rules of use 1-168
Relational operator
in Condition segment 1-644
segment 1-761
with WHERE keyword in SELECT 1-475
RELATIVE keyword
syntax in FETCH 1-296
use in FETCH 1-300
Release notes Intro-28
Remainder, in fragment expressions 1-39
RENAME COLUMN statement
restrictions 1-428
syntax 1-428
RENAME DATABASE statement 1-431
RENAME TABLE statement
ANSI-compliant naming 1-432
syntax 1-432
REPAIR TABLE statement, syntax and use 1-435
Repeatable Read isolation level
description of 1-558, 1-578
emulating during update 1-305
REPEATABLE READ keywords
syntax in SET ISOLATION 1-556
syntax in SET TRANSACTION 1-575
Reserved words 1-724
in ANSI SQL standard 1-724
use with delimited identifiers 1-727
use with identifiers 1-724
Resolution
in UPDATE STATISTICS 1-628, 1-629
with data distributions 1-628
RESOURCE keyword
use
in REVOKE 1-446
Resource privilege 1-343
Result of triggering statement 1-208
RETURN statement
returning insufficient values 2-41
returning null values 2-41
syntax 2-41
REVOKE FRAGMENT statement
syntax 1-450
REVOKE statement
column-specific privileges 1-445
database-level privileges 1-445
privileges needed 1-439
RESTRICT option 1-441
syntax 1-438
table-level privileges
ALL keyword 1-444
description 1-442
using with roles 1-440
Role
creating with CREATE ROLE statement 1-145
definition 1-145
dropping with DROP ROLE statement 1-271
enabling with SET ROLE 1-571

granting privileges with GRANT statement 1-345
revoking privileges with REVOKE 1-440
setting with SET ROLE 1-571
ROLLBACK WORK statement
syntax 1-455
use with WHENEVER 1-78, 1-86, 1-456
with DROP DATABASE 1-267
with DROP INDEX statement 1-269
with DROP PROCEDURE statement 1-270
with DROP SYNONYM statement 1-272
with DROP TABLE statement 1-274
with DROP TRIGGER statement 1-277
with DROP VIEW statement 1-279
ROLLFORWARD DATABASE statement
exclusive locking 1-457
syntax 1-457
ROOT function
syntax in expression 1-684
use in expression 1-688
ROUND function
syntax in expression 1-684
use in expression 1-688
Round-robin distribution scheme
defined with ALTER FRAGMENT 1-35
Row
deleting 1-252
engine response to locked row 1-562
inserting
through a view 1-372
with a cursor 1-373
multirow queries with FETCH 1-298
order, guaranteeing independence of 1-201
phantom 1-557, 1-578
retrieving with FETCH 1-301
rowid definition 1-301
updating through a view 1-613
writing buffered rows with FLUSH 1-308
ROW keyword
use in ALTER TABLE 1-75
use in CREATE TABLE 1-190
Rowid
adding
column with INIT clause 1-35
to fragmented table with CREATE TABLE 1-183
to fragmented tables with ALTER TABLE 1-75
dropping from fragmented tables 1-76
use in a column expression 1-675
use in fragmented tables 1-35
used as column name 1-732
ROWID keyword 1-675
Rule
arbitrary 1-36, 1-126, 1-188
hash 1-36, 1-126, 1-188
range 1-35, 1-126, 1-187
Rules for stored procedures 1-214

S

SCALE field
with GET DESCRIPTOR 1-318
with SET DESCRIPTOR 1-546
Schema. *See* Data model.
Scroll cursor
definition of 1-239
FETCH with 1-300
use of 1-240
SCROLL keyword
syntax in DECLARE 1-234
use in DECLARE 1-240
SECOND keyword
syntax
in DATETIME data type 1-669
in INTERVAL data type 1-743
use
as DATETIME field qualifier 1-746
as INTERVAL field qualifier 1-749
Segment
defined 1-640
relation to SPL statements 1-640
relation to SQL statements 1-640
Select cursor
definition of 1-237
opening 1-392
reopening 1-395
use of 1-238
SELECT keyword
ambiguous use as procedure variable 1-737
syntax
in GRANT 1-349
in REVOKE 1-442
use
in GRANT 1-350
in REVOKE 1-443
Select privilege
definition of 1-350
SELECT statement
aggregate functions in 1-709
as an argument to a stored procedure 2-6
associating with cursor with DECLARE 1-238
BETWEEN condition 1-476
column numbers 1-491
cursor for 1-238, 1-492, 1-493
describing returned values with DESCRIBE 1-255
FOR READ ONLY clause 1-493
FOR UPDATE clause 1-492
FROM Clause 1-472
GROUP BY clause 1-483
HAVING clause 1-485
IN condition 1-476
in FOR EACH ROW section 1-201
INTO clause with ESQL 1-467
INTO TEMP clause 1-495
IS NULL condition 1-476
joining tables in WHERE clause 1-480
LIKE or MATCHES condition 1-477
null values in the ORDER BY clause 1-490
ORDER BY clause 1-487
relational-operator condition 1-475

restrictions with INTO clause 1-405
ROWID keyword 1-675
SELECT clause 1-461
select numbers 1-491
singleton 1-468
subquery with WHERE keyword 1-475
syntax 1-459
UNION operator 1-498
use of expressions 1-463
with
DECLARE 1-234
FOREACH 2-23
INSERT 1-378
INTO keyword 1-302
LET 2-32
writing rows retrieved to an ASCII file 1-605
Self-join
description of 1-482
Sequential cursor
definition of 1-239
use of 1-239
with FETCH 1-299
SERIAL data type
in INSERT 1-377
resetting values 1-67
syntax 1-665
with stored procedures 2-10
Serializable isolation level
description of 1-578
SERIALIZABLE keyword, syntax in SET TRANSACTION 1-575
Server. *See* Database server.
Session control block
accessed by DBINFO function 1-693
defined 1-693
Session id
defined 1-693
returned by DBINFO function 1-693
SET clause 1-210
SET CONNECTION statement
CURRENT keyword 1-532
DEFAULT option 1-531
syntax and use 1-527
with concatenation operator 1-673
SET DATASKIP statement
syntax 1-534
what causes a skip 1-535
SET DEBUG FILE TO statement
syntax and use 1-537
with TRACE 2-46
SET DESCRIPTOR statement
syntax 1-540
the VALUE option 1-543
with concatenation operator 1-673
X/Open mode 1-545
SET EXPLAIN statement
interpreting output 1-549
MERGE JOIN information 1-550
optimizer access paths 1-549
output examples 1-551
SORT SCAN information 1-550
syntax 1-548
SET ISOLATION statement
default database levels 1-558
definition of isolation levels 1-557
effects of isolation 1-559
similarities to SET TRANSACTION statement 1-576
syntax 1-556
SET keyword
syntax in UPDATE 1-612
use in UPDATE 1-616
SET LOCK MODE statement
kernel locking 1-562
setting wait period 1-562
syntax 1-561
SET LOG statement
buffered vs. unbuffered 1-564
syntax 1-564
SET OPTIMIZATION statement, syntax and use 1-566
SET PDQPRIORITY statement 1-568
SET ROLE statement 1-571
SET SESSION AUTHORIZATION statement 1-573
SET statement
error options 1-504
list mode format 1-502
privileges required for executing 1-502
purpose 1-501
relationship to START VIOLATIONS TABLE 1-505, 1-585
relationship to STOP VIOLATIONS TABLE 1-507
syntax 1-501
table mode format 1-502
use
in setting transaction mode of constraints 1-523
with CREATE TRIGGER 1-217
with data definition statements 1-515
with data manipulation statements 1-509
with diagnostics tables 1-505
with violations tables 1-505
SET TRANSACTION statement
default database levels 1-579
definition of isolation levels 1-578
effects of isolation 1-580
similarities to SET ISOLATION statement 1-576
syntax 1-575
SHARE keyword, syntax in LOCK TABLE 1-387
Simple assignment 2-32
SIN function
syntax in expression 1-704
use in expression 1-705
Single-threaded application 1-529
Singleton SELECT statement 1-468
SITENAME function
returns server name 1-679
syntax
in expression 1-676
in INSERT 1-375
use
in ALTER TABLE 1-51
in CREATE TABLE 1-159
in expression 1-679
in INSERT 1-377
Slash (/), arithmetic operator 1-671
SMALLFLOAT data type
syntax 1-665
SMALLINT data type

syntax 1-665
using as default value 1-52, 1-160
SOME keyword
beginning a subquery 1-479
use in Condition subquery 1-655
Sorting
in a combined query 1-498
in SELECT 1-487
SPL
statements described 2-3
SQL
comments 1-9
compliance of statements with ANSI standard 1-16
keywords 1-724
statement types 1-12
SQL Communications Area (SQLCA)
and EXECUTE statement 1-283
result after CLOSE 1-82
result after DATABASE 1-229
result after DESCRIBE 1-256
result after FETCH 1-306
result after FLUSH 1-309
result after OPEN 1-392, 1-393
result after PUT 1-422
result after SELECT 1-471
warning when dbspace skipped 1-534
SQL statement. *See*Statement, SQL .
SQLCA. *See* SQL Communications Area.
sqlda structure
syntax
in DESCRIBE 1-256
in FETCH 1-296
in OPEN 1-284, 1-287, 1-391, 1-417
use
in DESCRIBE 1-257
in FETCH 1-304
in OPEN 1-398
in PUT 1-421
use with EXECUTE statement 1-287
SQLERROR keyword, in the WHENEVER statement 1-632, 1-635
SQLNOTFOUND, error conditions with EXECUTE statement 1-289
SQLSTATE
Not Found condition 1-637
runtime errors 1-635
warnings 1-636
SQLSTATE variable
list of codes 1-323
SQLWARNING keyword, in the WHENEVER statement 1-632, 1-636
SQRT function
syntax in expression 1-684
use in expression 1-689
START DATABASE statement
syntax and use 1-581
START VIOLATIONS TABLE statement
description of 1-584
privileges required for executing 1-587
relationship to SET 1-505, 1-585
relationship to STOP VIOLATIONS TABLE 1-586
syntax 1-584
Statement
SQL
Statements that are extensions to ANSI standard 1-16
Statement identifier
associating with cursor 1-238
definition of 1-403
releasing 1-404
syntax
in DECLARE 1-234
in FREE 1-311
use
in DECLARE 1-247
in FREE 1-313
in PREPARE 1-403
Statement, SQL
ANSI-compliant 1-16
ANSI-compliant with Informix extensions 1-16
how to enter 1-6
types 1-12
STATUS FOR keywords, in INFO statement 1-369
Status, displaying with INFO statement 1-369
STDEV function 1-718
STOP keyword, in the WHENEVER statement 1-632, 1-637
STOP VIOLATIONS TABLE statement
description of 1-603
privileges required for executing 1-604
relationship to SET 1-507
relationship to START VIOLATIONS TABLE 1-586
syntax 1-603
Stored procedure
as triggered action 1-214
BYTE and TEXT data types 2-15
checking references 1-214
comments in 1-10, 1-138
cursors with 2-24
DBA-privileged, how to create 1-135
DBA-privileged, use with triggers 1-215
debugging 2-46
definition of 2-3
displaying documentation 1-139
executing operating system commands from 2-43
granting privileges on 1-345
handling multiple rows 2-42
header 2-9
in SELECT statements 1-465
in WHEN condition 1-207
naming output file for TRACE statement 1-537
owner-privileged 1-135, 1-215
privileges 1-215
receiving data from SELECT 1-467
revoking privileges on 1-439
sending mail from 2-44
simulating errors 2-39
stores7 database
copying Intro-9
creating Intro-9
overview Intro-8
See also Demonstration database.

Structured Query Language. *See* SQL.
Subquery
  beginning with ALL / ANY / SOME keywords 1-479
  beginning with EXISTS keyword 1-478
  beginning with IN keyword 1-478
  correlated 1-653
  definition of 1-475
  in Condition segment 1-652
  restrictions with UNION operator 1-499
  with DISTINCT keyword 1-463
Subscripting
  on character columns 1-489, 1-675
Substring
  in column expression 1-675
  in ORDER BY clause of SELECT 1-489
SUM function
  syntax in expression 1-709
  use in expression 1-713
Surviving
  index 1-22
  table 1-22, 1-25
Synonym
  ANSI-compliant naming 1-150
  chains 1-152
  creating with CREATE SYNONYM 1-150
  difference from alias 1-150
  dropping 1-272
  naming conventions 1-767
Synonym Name segment
  syntax 1-766
  use 1-767
syscolauth system catalog table 1-443
sysdepend system catalog table 1-279
systabauth system catalog table 1-443
System catalog
  database entries 1-105
  syscolauth 1-443
  systabauth 1-354, 1-443
System descriptor area
  assigning values to 1-542
  modifying contents 1-543
  resizing 1-543
  use of 1-258
  use with EXECUTE statement 1-288
System name, in database name 1-662
SYSTEM statement
  setting environment variables with 2-44
  syntax 2-43

T

Table
  adding a constraint 1-68, 1-69, 1-70
  algorithms for adding columns 1-49
  alias in SELECT 1-472
  ANSI-compliant naming 1-770
  checking with the CHECK TABLE statement 1-79
  consumed 1-25
  creating
    a synonym for 1-150
    a table 1-154
    a temporary table 1-177
  defining fragmentation strategy 1-186
  defining temporary 1-180
  diagnostic 1-596
  dropping
    a synonym 1-272
    a table 1-274
  dropping a constraint 1-73
  engine response to locked table 1-562
  fragmenting 1-183
  joins in Condition segment 1-480
  loading data with the LOAD statement 1-380
  locking
    changing mode 1-75
    with ALTER INDEX 1-44
    with LOCK TABLE 1-387
  logging 1-182
  naming conventions 1-154, 1-770
  optimizing queries 1-626
  repairing with REPAIR TABLE statement 1-435
  restoring with audit trail 1-425
  storing in dbspaces 1-183
  surviving 1-22, 1-25
  target 1-588
  temporary 1-26
  unlocking 1-610
  violations 1-587
TABLE keyword, syntax in UPDATE STATISTICS 1-624
Table mode format, in SET 1-502
Table Name segment 1-768
Table-level privilege
  column-specific privileges 1-445
  default with GRANT 1-352
  definition and use 1-350
  granting 1-349
  passing grant ability 1-353
  revoking 1-442
TABLES keyword, in INFO statement 1-366
tabtype 1-108
TAN function
  syntax in expression 1-704
  use in expression 1-705
Target table
  relationship to diagnostics table 1-588
  relationship to violations table 1-588
TEMP keyword
  syntax in SELECT 1-459
  use in SELECT 1-495
TEMP TABLE keywords, syntax in CREATE TABLE 1-154
Temporary table
  building distributions 1-631
  creating constraints for 1-180
  DBSPACETEMP environment variable 1-178
  defining columns 1-180
  explicit 1-177
  implicit 1-178
  naming 1-178
  storage

created with CREATE TABLE 1-178
created with SELECT INTO TEMP 1-496
updating statistics 1-626
when deleted 1-177
TEXT data type
requirements for LOAD statement 1-384
syntax 1-665
with stored procedures 2-10, 2-15
Thread
defined 1-529
in multithreaded application 1-529
Thread-safe application
description 1-263, 1-529, 1-532
Time function
restrictions with GROUP BY 1-484
use in SELECT 1-464
TO CLUSTER keywords, in ALTER INDEX 1-44
TO keyword
in expression 1-709
in GRANT 1-340
TODAY function
syntax
in Condition segment 1-644
in expression 1-676
in INSERT 1-375
use
in ALTER TABLE 1-51
in constant expression 1-680
in CREATE TABLE 1-159
in INSERT 1-377
TP/XA. *See* Transaction manager.
TRACE statement
syntax 2-46
Transaction
and CREATE DATABASE 1-107
in active connection 1-93
recovering transactions 1-457
rolling back 1-455
starting with BEGIN WORK 1-77
stopping logging 1-581
using cursors in 1-248
Transaction logging
description of 1-581
renaming log 1-582
stopping 1-582
Transaction mode, for constraints 1-523
Trigger
affected by dropping a column from table 1-65
effects on altered fragments 1-29
effects with ALTER FRAGMENT ATTACH 1-26
in client/server environment 1-219
number on a table 1-194
object modes for 1-222
setting with SET 1-501
preventing overriding 1-219
Trigger event
definition of 1-194
in CREATE TRIGGER statement 1-194
INSERT 1-202
privileges on 1-195
with cursor statement 1-195
Trigger name
syntax 1-196
Triggered action
action on triggering table 1-212
anyone can use 1-216
cascading 1-201
clause
action statements 1-207
WHEN condition 1-206
correlation name in 1-210, 1-214
in client/server environment 1-219
list
AFTER 1-200
BEFORE 1-199
FOR EACH ROW 1-200
for multiple triggers 1-200
sequence of 1-199
merged 1-200
preventing overriding 1-219
syntax 1-206
WHEN condition 1-206
Triggering statement
affecting multiple rows 1-201
execution of 1-195
guaranteeing same result 1-194
result of 1-208
UPDATE 1-198
Triggering table
action on 1-212
and cascading triggers 1-217
Trigonometric function
ACOS function 1-706
ASIN function 1-706
ATAN function 1-706
ATAN2 function 1-706
COS function 1-705
SIN function 1-705
TAN function 1-705
TRIM function 1-707
TRUNC function
syntax in expression 1-684
use in expression 1-690
TYPE field
changing from BYTE or TEXT 1-547
setting in SET DESCRIPTOR 1-544
setting in X/Open programs 1-545
with X/Open programs 1-317

U

Underscore (_), wildcard in Condition segment 1-650
UNION operator
restrictions on use 1-498
syntax in SELECT 1-459
use in SELECT 1-498
Unique constraint
dropping 1-73, 1-74
modifying a column with 1-67
rules of use 1-71, 1-167, 1-176
UNIQUE keyword
syntax
in CREATE INDEX 1-109
in CREATE TABLE 1-162
in SELECT 1-461
use
in ALTER TABLE 1-69
in CREATE INDEX 1-110
in expression 1-709
in SELECT 1-463

no effect in subquery 1-654
UNITS keyword
syntax in expression 1-676
use in expression 1-682
UNLOAD statement
DELIMITER clause 1-608
syntax 1-605
UNLOAD TO file 1-606
unloading VARCHAR, TEXT, or BYTE columns 1-607
UNLOAD TO file 1-606
UNLOCK TABLE statement, syntax and use 1-610
Updatable view 1-228
UPDATE clause, syntax 1-197
Update cursor
definition of 1-237
locking considerations 1-243
opening 1-392
restricted statements 1-245
use in UPDATE 1-620
using 1-242
UPDATE keyword
syntax
in GRANT 1-349
in REVOKE 1-442
use
in GRANT 1-350
in REVOKE 1-443
Update privilege
definition of 1-350
with a view 1-613
UPDATE REFERENCING clause
and FOR EACH ROW section 1-205
syntax 1-204
UPDATE statement
and transactions 1-614
as triggered action 1-207
as triggering statement 1-195, 1-197, 1-198
in trigger event 1-194
locking considerations 1-615
restrictions on columns for update 1-243
rolling back updates 1-614
syntax 1-613
updating a column to null 1-618
updating through a view 1-613
updating with cursor 1-620
use of expressions 1-619
with
a SELECT...FOR UDATE 1-492
FETCH 1-305
SET keyword 1-616
WHERE CURRENT OF keywords 1-620
WHERE keyword 1-620
with an update cursor 1-242
UPDATE STATISTICS statement
and temporary tables 1-631
creating data distributions 1-628
dropping data distributions 1-628
examining index pages 1-626
optimizing search strategies 1-626, 1-631
recommended procedure for using 1-631
specifying distributions only 1-630
syntax 1-623
using the LOW keyword 1-627
when to execute 1-627
Update trigger, defining multiple 1-197
Upgrading the database server 1-627
Upgrading to a newer database server 1-627
USER function
as affected by ANSI compliance 1-439, 1-679
syntax
in Condition segment 1-644
in expression 1-676
in INSERT 1-375
use
in ALTER TABLE 1-51
in CREATE TABLE 1-159
in expression 1-678
in INSERT 1-377
User informix, privileges associated with 1-344
Using correlation names 1-209
USING DESCRIPTOR keywords
information from DESCRIBE 1-258
syntax
in EXECUTE 1-281
in FETCH 1-296
in OPEN 1-390
in PUT 1-416
use
in FETCH 1-304
in OPEN 1-398
in PUT 1-289, 1-421
USING keyword
syntax
in EXECUTE 1-286
in OPEN 1-390
use
in EXECUTE 1-286
in OPEN 1-396
USING SQL DESCRIPTOR keywords
in DESCRIBE 1-258
in EXECUTE 1-288

V

VALUE clause
after NULL value is fetched 1-319
relation to FETCH 1-318
use in GET DESCRIPTOR 1-317
use in SET DESCRIPTOR 1-543
VALUES clause
effect with PUT 1-419
syntax in INSERT 1-370
use in INSERT 1-375
VARCHAR data type
considerations for UNLOAD statement 1-607
requirements for LOAD statement 1-384
syntax 1-665
using as default value 1-52, 1-160
Variable
default values in SPL 2-12, 2-13
define in SPL 2-8
global, in SPL 2-11
local, in SPL 2-13
scope of SPL variable 2-9
unknown values in IF 2-28
VARIANCE function 1-718
View

affected by dropping a column from base table 1-65
creating a view 1-224
creating synonym for 1-150
dropping 1-279
privilege when creating 1-225
privileges with GRANT 1-355
restrictions with UNION operator 1-499
updatable 1-228
updating 1-613
virtual column 1-226
with SELECT * notation 1-224
View Name segment 1-772
Violations table
creating with START VIOLATIONS TABLE 1-584
examples 1-510, 1-516, 1-592, 1-595, 1-604
how to start 1-506, 1-584
how to stop 1-507, 1-603
privileges on 1-590
relationship to diagnostics table 1-588
relationship to target table 1-588
restriction on dropping 1-275
structure 1-587
use with SET 1-505
when to start 1-506

W

WAIT keyword, in the SET LOCK MODE statement 1-561
Warning
if dbspace skipped 1-534
WEEKDAY function
syntax in expression 1-699
use in expression 1-701
WHEN condition
in triggered action 1-207
restrictions 1-207
use of 1-207
WHENEVER statement, syntax and use 1-632
WHERE CURRENT OF keywords
impact on trigger 1-195
syntax
in DELETE 1-252
in UPDATE 1-612
use
in UPDATE 1-620
WHERE keyword
joining tables 1-480
setting descriptions of items 1-542
syntax
in DELETE 1-252
in SELECT 1-459
in UPDATE 1-612
use
in DELETE 1-253
in UPDATE 1-620
with a subquery 1-475
with ALL keyword 1-479
with ANY keyword 1-479
with BETWEEN keyword 1-476
with IN keyword 1-476
with IS keyword 1-476
with LIKE keyword 1-477
with MATCHES keyword 1-477
with relational operator 1-475
with SOME keyword 1-479
WHILE keyword
in CONTINUE statement 2-7
in EXIT 2-16
WHILE statement
syntax 2-49
with NULL expressions 2-49
Wildcard character
asterisk (*) 1-651
backslash (› as escape character 1-650, 1-651
caret (^) 1-651
percent sign (%) 1-650
question mark (?) 1-651
underscore (_) 1-650
with LIKE or MATCHES 1-477, 1-649, 1-759
wildcard in Condition segment 1-651
WITH APPEND keywords, in the SET DEBUG FILE TO statement 1-537
WITH CHECK keywords
syntax in CREATE VIEW 1-224
use in CREATE VIEW 1-227
WITH GRANT keywords
syntax in GRANT 1-340
use in GRANT 1-353
WITH HOLD keywords
syntax in DECLARE 1-234
use in DECLARE 1-240, 1-251
WITH keyword, syntax in CREATE DATABASE 1-104
WITH LOG IN keywords, syntax in START DATABASE 1-581
WITH NO LOG keywords
syntax
in CREATE TABLE 1-154
in SELECT 1-495
in START DATABASE 1-581
use
in CREATE TABLE 1-182
in SELECT 1-498
in START DATABASE 1-582
WITH RESUME keywords, in RETURN 2-42
WITH ROWIDS clause, of CREATE TABLE 1-183
WITHOUT HEADINGS keywords, in the OUTPUT statement 1-401

X

X/Open
specifications, icon for Intro-19
X/Open mode
FETCH statement 1-299
GET DESCRIPTOR 1-317
SET DESCRIPTOR statement 1-545

Y

YEAR function
syntax in expression 1-699
use in expression 1-702
YEAR keyword
syntax
in DATETIME data type 1-669
in INTERVAL data type 1-743
use
as DATETIME field qualifier 1-746

as INTERVAL field
qualifier 1-749

Symbols

", double quotes
around delimited identifier 1-726
around quoted string 1-757
within delimited identifier 1-729
within quoted string 1-759
$INFORMIXDIR/etc/sqlhosts. *See* sqlhosts file.
%, percent sign, wildcard in Condition segment 1-650
) 1-651
*, asterisk
wildcard in Condition segment 1-651
**, asterisk
arithmetic operator 1-671
use in SELECT 1-461
+, plus sign, arithmetic operator 1-671
--, double dash, comment symbol 1-9
-, minus sign, arithmetic operator 1-671
, square brackets, wildcard in Condition segment 1-651
., period 1-135
1-651
Square brackets 1-651
Wildcard character
square brackets (1-651
/, division symbol, arithmetic operator 1-671
/, slash, arithmetic operator 1-671
?, question mark
as placeholder in PREPARE 1-403
naming variables in PUT 1-420
replacing with USING keyword 1-396
wildcard in Condition segment 1-651
\, backslash, as escape character
with LIKE 1-650
with MATCHES 1-651
^, caret, wildcard in Condition segment 1-651
_, underscore, wildcard in Condition segment 1-650
{}, curly brackets, comment symbol 1-9
||, concatenation operator 1-672
', single quotes
around quoted string 1-757
within quoted string 1-759